LASCAR : un langage pour la simulation et l'évaluation des architectures d'ordinateurs

Dominique Borrione

THESE présentée à

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

POUR OBTENIR LE GRADE DE
DOCTEUR DE 3ème CYCLE
Spécialité Informatique

Dominique BORRIONE

LASCAR

UN LANGAGE POUR LA SIMULATION ET L'EVALUATION DES ARCHITECTURES D'ORDINATEURS

Thèse soutenue le 16 avril 1976 devant la Commission d'Examen

Président : L. BOLLIET

Examinateurs : F. ANCEAU
S. KRAKOWIAK
J. MERMET

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Monsieur Michel SOUTIF : Président

Monsieur Gabriel CAU : Vice-Président

MEMBRES DU CORPS ENSEIGNANTS DE L'U.S.M.G.

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des Fluides
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique Approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique Expérimentale
	BARBIER Reynold	Géologie Appliquée
	BARJON Robert	Physique Nucléaire
	BARNOUD Fernand	Biosynthèse de la Cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique Chirurgicale
	BEAUDOING André	Clinique de Pédiatrie et Puériculture
	BERNARD Alain	Mathématiques Pures
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BERTRANDIAS Jean-Paul	Mathématiques Pures
	BEZES Henri	Pathologie Chirurgicale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (I.U.T. B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique Ophtalmologique
	BONNET-EYMARD Joseph	Clinique Gastro-entérologique
Mme	BONNIER Marie-Jeanne	Chimie Générale
MM.	BOUCHERLE André	Chimie et Toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques Appliquées
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique Rhumatologique et Hydrologique
	CALAS François	Anatomie
	CARLIER Georges	Biologie Végétale
	CARRAZ Gilbert	Biologie Animale et Pharmacodynamie
	CAU Gabriel	Médecine Légale et Toxicologie
	CAUQUIS Georges	Chimie Organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Clinique Oto-Rhino-Laryngologique
	CHATEAU Robert	Clinique de Neurologie
	CHIBON Pierre	Biologie Animale
	COEUR André	Pharmacie Chimique et Chimie Analytique
	CONTAMIN Robert	Clinique Gynécologique
	COUDERC Pierre	Anatomie Pathologique
	CRAYA Antoine	Mécanique
Mme	DEBELMAS Anne-Marie	Matière Médicale
MM.	DEBELMAS Jacques	Géologie Générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumo-Phtisiologie
	DEPORTES Charles	Chimie Minérale
	DESRE Pierre	Métallurgie

MM.	DESSAUX Georges	Physiologie Animale
	DODU Jacques	Mécanique Appliquée (I.U.T. A)
	DOLIQUE Jean-Michel	Physique des Plasmas
	DREYFUS Bernard	Thermodynamique
	DUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de Dermatologie et Syphiligraphie
	GAGNAIRE Didier	Chimie Physique
	GALLISSOT François	Mathématiques Pures
	GALVANI Octave	Mathématiques Pures
	GASTINEL Noël	Analyse Numérique
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques Pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique Générale
	KLEIN Joseph	Mathématiques Pures
	KOSZUL Jean-Louis	Mathématiques Pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques Appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie Végétale
Mme	LAJZEROWICZ Janine	Physique
MM.	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie Générale
	LATURAZE Jean	Biochimie Pharmaceutique
	LAURENT Pierre-Jean	Mathématiques Appliquées
	LEDRU Jean	Clinique Médicale B
	LLIBOUTRY Louis	Géophysique
	LOISEAUX Pierre	Sciences Nucléaires
	LONGEQUEUE Jean-Pierre	Physique Nucléaires
	LOUP Jean	Géographie
Melle	LUTZ Elisabeth	Mathématiques Pures
	MALGRANGE Bernard	Mathématiques Pures
	BOUTET DE MONVEL Louis	Mathématiques Pures
	MALINAS Yves	Clinique Obstétricale
	MARTIN-NOEL Pierre	Seméiologie médicale
	MAZARE Yves	Clinique Médicale A
	MICHEL Robert	Minéralogie et Pétrographie
	MICOUD Max	Clinique Maladies Infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie Nucléaire
	MULLER Jean-Michel	Thérapeutique (Néphrologie)
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques Pures
	PEBAY-PEYROULA Jean-Claude	Physique
	RASSAT André	Chimie Systématique
	RENARD Michel	Thermodynamique
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-Chirurgie
	SEIGNEURIN Raymond	Microbiologie et Hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction Mécanique (I.U.T. A)
	SOUTIF Michel	Physique Générale
	TANCHE Maurice	Physiologie

MM.	TRAYNARD Philippe	Chimie Générale
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique Nucléaire
	VAUQUOIS Bernard	Calcul Electronique
Mme	VERAIN Alice	Pharmacie Galénique
MM.	VERAIN André	Physique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique Nucléaire Théorique

PROFESSEURS ASSOCIES

MM.	CLARK Gilbert	Spectrométrie Physique
	CRABBE Pierre	CERMO
	ENGLMAN Robert	Spectrométrie Physique
	HOLTZBERG Frédéric	Basses Températures
	ROST Ernest	Sciences Nucléaires

PROFESSEURS SANS CHAIRE

Melle	AGNIUS-DELORD Claudine	Physique Pharmaceutique
	ALARY Josette	Chimie Analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques Appliquées
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique (I.U.T. A)
	BUISSON René	Physique (I.U.T. A)
	CONTE René	Physique (I.U.T. A)
	DEPASSEL Roger	Mécanique des Fluides
	GAUTHIER Yves	Sciences Biologiques
	GAUTRON René	Chimie
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie Médicale
	HACQUES Gérard	Calcul Numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Médecine Préventive
	IDELMAN Simon	Physiologie Animale
	JOLY Jean-René	Mathématiques Pures
	JULLIEN Pierre	Mathématiques Appliquées
Mme	KAHANE Josette	Physique
MM.	KUHN Gérard	Physique (I.U.T. A)
	LE ROY Philippe	Mécanique (I.U.T. A)
	LUU DUC Cuong	Chimie Organique
	MAYNARD Roger	Physique du Solide
	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et Minéralogie
	PSISTER Jean-Claude	Physique du Solide
Melle	PIERY Yvette	Physiologie Animale
MM.	RAYNAUD Hervé	M.I.A.G.
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie Générale
	RICHARD Lucien	Biologie Végétale
Mme	RINAUDO Marguerite	Chimie Macromoléculaire
MM.	ROBERT André	Chimie Papetière

MM.	SARRAZIN Roger	Anatomie et Chirurgie
	SARROT-REYNAUD Jean	Géologie
	SIROT Louis	Chirurgie Générale
Mme	SOUTIF Jeanne	Physique Générale
MM.	STREGLITZ Paul	Anesthésiologie
	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques Appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	AMBLARD Pierre	Dermatologie
	ARMAND Gilbert	Géographie
	ARMAND Yves	Chimie (I.U.T. A)
	BACHELOT Yvan	Endocrinologie
	BARGE Michel	Neuro chirurgie
	BARJOLLE Michel	MIAG
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamie
MM.	BOST Michel	Pédiatrie
	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (C.U.S.)
MM.	BRODEAU François	Mathématiques (I.U.T. B)
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et Organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie Papetière
	CHIAVERINA Jean	Biologie Appliquée (EFP)
	COHEN-ADDAD Jean-Pierre	Spectrométrie Physique
	COLOMB Maurice	Biochimie Médicale
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	CORDONNIER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	CYROT Michel	Physique du solide
	DELOBEL Claude	M.I.A.G.
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie Végétale
	DUSSAUD René	Mathématiques (C.U.S.)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine Légale
	FAURE Gilbert	Urologie
	FONTAINE Jean-Marc	Mathématiques Pures
	GAUTIER Robert	Chirurgie Générale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GROS Yves	Physique (I.U.T. A)
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	KOLODIE Lucien	Hématologie
	KRAKOWIAK Sacha	Mathématiques Appliquées
	LE NOC Pierre	Bactériologie-virologie
	LEROY Philippe	I.U.T. A
	MACHE Régis	Physiologie Végétale
	MAGNIN Robert	Hygiène et Médecine Préventive
	MALLION Jean-Michel	Médecine du Travail
	MARECHAL Jean	Mécanique (I.U.T. A)
	MARTIN-BOUYER Michel	Chimie (C.U.S.)

M.	MICHOULIER Jean	Physique (I.U.T. A)
Mme	MINIER Colette	Physique (I.U.T. A)
MM.	NEGRE Robert	Mécanique (I.U.T. A)
	NEMOZ Alain	Thermodynamique
	NOUGARET Marcel	Automatique (I.U.T.A)
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (I.U.T. B)
	PEFFEN René	Métallurgie (I.U.T. A)
	PERRET Jean	Neurologie
	PERRIER Guy	Géophysique - Glaciologie
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine Interne
	RACINET Claude	Gynécologie et Obstétrique
	RAMBAUD André	Hygiène et Hydrologie
	RAMBAUD Pierre	Pédiatrie
Mme	RENAUDET Jacqueline	Bactériologie
MM.	ROBERT Jean-Bernard	Chimie-Physique
	ROMIER Guy	Mathématiques (I.U.T. B)
	SHOM Jean-Claude	Chimie générale
	STOEBNER Pierre	Anatomie pathologique
	VROUSOS Constantin	Radiologie

MAITRE DE CONFERENCES ASSOCIES

M.	COLE Antony	Sciences Nucléaires

CHARGE DE FONCTIONS DE MAITRE DE CONFERENCES

M.	JUNIEN-LAVILLAVROY Paul	O.R.L.

Fait à SAINT MARTIN D'HERES,
DECEMBRE 1975.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : M. NEEL Louis

Vice-Présidents : M. BENOIT Jean

M. BONNETAIN Lucien

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Radioélectricité
	BESSON Jean	Electrochimie
	BLOCH Daniel	Physique du solide
	BONNETAIN Lucien	Chimie Minérale
	BONNIER Etienne	Electrochimie et Electrométallurgie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	Electronique
	COUMES André	Radioélectricité
	FELICI Noël	Electrostatique
	LESPINARD Georges	Mécanique
	MOREAU René	Mécanique
	PARIAUD Jean-Charles	Chimie-Physique
	PAUTHENET René	Physique du solide
	PERRET René	Servomécanismes
	POLOUJADOFF Michel	Electrotechnique
	SILBERT Robert	Mécanique des Fluides

PROFESSEURS ASSOCIES

MM.	RUPPERSBERG Albert, Henner	Chimie
	ROUXEL Roland	Automatique

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuel	Electronique
	BOUVARD Maurice	Génie Mécanique
	COHEN Joseph	Electrotechnique
	DURAND Francis	Métallurgie
	FOULARD Claude	Automatique
	LACOUME Jean-Louis	Géophysique
	LANCIA Roland	Electronique
	VEILLON Gérard	Informatique Fondamentale & Appliquée
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM.	ANCEAU François	Mathématiques Appliquées
	BOUDOURIS Georges	Radioélectricité
	CHARTIER Germain	Electronique
	GUYOT Pierre	Chimie Minérale
	IVANES Marcel	Electrotechnique
	JOUBERT Jean-Claude	Physique du solide
	MORET Roger	Electrotechnique Nucléaire
	PIERRARD Jean-Marie	Hydraulique
	ROBERT François	Analyse Numérique
	SABONNADIERE Jean-Claude	Informatique Fondamentale & Appliquée
Mme	SAUCIER Gabrièle	Informatique Fondamentale & Appliquée

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan	Automatique

CHERCHEURS DU C.N.R.S. (Directeurs et Maîtres de Recherche)

MM.	FRUCHART Robert	Directeur de Recherche
	ANSARA Ibrahim	Maître de Recherche
	CARRE René	Maître de Recherche
	DRIOLE Jean	Maître de Recherche
	MATHIEU Jean-Claude	Maître de Recherche
	MUNIER Jacques	Maître de Recherche

Je tiens à remercier

Monsieur le Professeur Louis BOLLIET, Directeur du département informatique de l'Institut Universitaire de Technologie de Grenoble, qui m'a toujours encouragée dans mon travail, et qui m'a fait l'honneur de présider le jury de cette thèse ;

Monsieur Jean MERMET, Chargé de Recherches au C.N.R.S. qui m'a accueillie dans son équipe, et qui a su à la fois me conseiller dans mon travail et me laisser une grande autonomie pour le mener à bien ;

Monsieur François ANCEAU, Maître de Conférences à l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble, et Monsieur Sacha KRAKOWIAK, Maître de Conférences à l'Université Scientifique et Médicale de Grenoble, qui, par leurs critiques bienveillantes et constructives, ont largement contribué à l'amélioration de cet ouvrage.

Je voudrais remercier aussi les personnes qui ont défini le projet multi-processeur, Madame RECOQUE et Messieurs CENSIER et MAZARE ; au cours des réunions de travail, sympathiques et animées, qui nous ont rassemblés, j'ai beaucoup appris sur la structure, matérielle et logicielle, des systèmes informatiques.

Je ne saurais trop souligner l'apport essentiel des membres de l'Equipe CASSANDRE, que je n'ai pas tous connus, et dont le travail est à la base de la mise en oeuvre du langage LASCAR.

Sans la contribution de mes deux collègues les plus proches, ce travail n'aurait sans doute jamais été mené à terme. Monsieur Yvon BRESSY n'a jamais ménagé son temps pour m'expliquer le fonctionnement du système CASSANDRE, et m'a donné, par sa rigueur et son exemple, une méthode de travail que je ne possédais pas en programmation. Monsieur Jean-François GRABOWIECKI m'a apporté une aide considérable dans la réalisation technique de ce travail ; sa compétence et sa gentillesse m'ont été un encouragement constant. Je tiens à leur exprimer ici mon amitié sincère et ma profonde reconnaissance.

Enfin, je voudrais remercier

Madame Jacqueline BRICHET, qui a assuré la dactylographie de cette thèse avec beaucoup de soin et de goût ;

Le Service de Reprographie du Centre Inter-Universitaire de Calcul de Grenoble, qui a réalisé avec compétence le tirage de cet ouvrage.

A mes Parents

SOMMAIRE

INTRODUCTION

I . LES PRINCIPAUX TYPES DE MODÈLES

I.1 LES MODELES ANALYTIQUES
I.2 LES MODELES DE GRAPHES DE CONTROLE
I.3 LES MODELES DESCRIPTIFS
I.4 LES MODELES DE SIMULATION

II . LES OUTILS D'AIDE À LA CONCEPTION

II.1 LES MODELES ABSTRAITS
II.2 LES MODELES DE DESCRIPTION DE MATERIEL
II.3 NECESSITE D'UN SYSTEME INTEGRE D'AIDE A LA CONCEPTION

PREMIERE PARTIE

DÉFINITION DU LANGAGE LASCAR

DEUXIEME PARTIE

UN EXEMPLE D'APPLICATION : MODÉLISATION D'UNE STRUCTURE MULTI-PROCESSEUR

TROISIEME PARTIE

PROGRAMMES DE TRAITEMENT
EVALUATION

INTRODUCTION

La complexité croissante des systèmes informatiques et de leurs applications a rendu indispensable l'élaboration de techniques de modélisation, comme aides à la conception et à l'amélioration de ces systèmes. Un modèle, dans son acception la plus générale, est une représentation du phénomène réel qui permet soit d'en extraire certaines caractéristiques soit de se substituer à ce phénomène pour une application particulière. Nous ne faisons donc à priori aucune hypothèse sur le degré de précision de la représentation, étant entendu que nous ne parlerons ici que de modèles de systèmes informatiques.

L'élaboration de modèles s'est révélée utile à tous les stades de développement et d'utilisation d'un système informatique, et a répondu à des besoins très variés.

Documentation

Au niveau de la conception d'une nouvelle structure de machine, un modèle descriptif des différents composants et de leurs inter-connections peut servir de moyen de communication entre équipes de spécialités complémentaires, et même entre les membres d'une même équipe. Ces descriptions ont longtemps été sous forme d'équations logiques ou de plans. Depuis quelques années, on s'oriente vers l'utilisation de langages formels de description, où les composants d'un système sont réunis dans une base de donnée [A7].

Vérification de bon fonctionnement

Lorsqu'un composant est structurellement et fonctionnellement figé, le coût de fabrication d'un prototype impose que son bon fonctionnement logique ait été entièrement vérifié au préalable. L'outil habituellement utilisé par les constructeurs est la simulation [D6], [D13], [D17].

Prédiction de performance

Le problème majeur de la prédiction de performance d'un nouveau système ou de l'amélioration d'un système existant est devenu extrêmement complexe, de part le nombre de facteurs qui influent que cette performance, tant au niveau matériel que logiciel. Ce vaste sujet a donné lieu à de très nombreuses études, et a été appréhendé soit par des modèles analytiques, soit par des modèles de simulation.

Aide à la compréhension

La modélisation d'un système opérationnel permet une meilleure compréhension de son fonctionnement, et outre son caractère explicatif a un intérêt pédagogique évident. Minsky, à cet égard, donne de la modélisation une définition extrêmement large [F9] :

> "Pour un observateur A, Beta est un modèle de B si A peut, à partir de Beta, apprendre quelque chose d'utile sur B".

Examinons tout d'abord les principaux modèles qui sont couramment utilisés pour représenter les systèmes informatiques. Nous ne prétendons pas en dresser une liste exhaustive, mais plutôt éclairer par quelques exemples l'angle sous lequel ont été abordées les réponses les plus usuelles aux besoins que nous venons d'énumérer.

Pour en clarifier la présentation, nous les répartirons en quatre classes distinctes ; mais il est bien évident qu'en réalité ces classes ne sont pas disjointes, et que leur frontière peut être assez floue.

I . LES PRINCIPAUX MODELES

I.1 LES MODELES ANALYTIQUES

Depuis 1965, des études théoriques ont été entreprises pour dégager les principaux paramètres qui influent sur la performance des systèmes informatiques. Ces travaux ont débouché sur l'élaboration de modèles mathématiques, pour lesquels les comportements des ressources du système, et des programmes y faisant appel, ne sont connus que par des lois de probabilité. La théorie des processus stochastiques, ainsi que l'approximation des processus stochastiques par des processus de diffusion, est à la base de la résolution de nombreux modèles [C25] dont on considère le fonctionnement à l'état stationnaire [C15], [E3].

Les modèles globaux font généralement ressortir comme critère de performance soit la capacité de traitement (en nombre d'instructions par unités de temps), soit le taux d'utilisation des différentes ressources (mémoire principale, unité centrale et périphériques rapides) [C3], [C16], [C26].

Plus nombreux sont les modèles partiels de gestion d'une ressource particulière. Citons notamment :

- l'allocation de l'unité centrale, pour les systèmes à temps partagé. Le critère de performances est alors le temps de réponse, en fonction de l'attribution de quanta de temps et de priorités [C9], [C18], [C19].

- L'attribution d'ensembles de travail, d'après des études sur le comportement des programmes [C7], [C8], [C20] , et le choix d'algorithmes de remplacement de pages [C1], [C13] dans un système à mémoire virtuelle paginée.

- Le choix d'une hiérarchie de mémoire : technologie, taille des différents niveaux, taille des unités d'information transmises d'un niveau à l'autre (pages de la mémoire centrale, blocs d'une mémoire cache), algorithmes de remplacement des pages et des blocs [D2], [C29], [F9], [F22].

- L'optimisation de l'allocation des fichiers sur un disque, et le choix d'un algorithme de prise en compte des requêtes en fonction de l'emplacement des enregistrements [C4], [C12], [C27], [C28].

Les modèles analytiques, par la réflexion théorique que leur élaboration nécessite, ont permis de mieux dégager les concepts de fonctionnement communs aux principaux systèmes (notions de serveur, de ressource etc...) ; ceux-ci, en effet, sont souvent mal perçus, noyés dans les détails de mise en oeuvre en dessous d'un certain niveau d'abstraction.

Un autre avantage des modèles analytiques est de donner directement le résultat soit par des formules mathématiques, soit en utilisant des méthodes de résolution numérique. Une telle approche est donc peu coûteuse en temps machine.

L'emploi de modèles analytiques pose cependant des difficultés non négligeables :

. le fait qu'on ne sache les résoudre que pour certaines lois de probabilité (gaussiennes, ou combinaison linéaire d'exponentielles négatives) impose parfois des hypothèses trop restrictives par rapport au phénomène réel.

. le problème important de la validation des résultats n'est pas toujours soluble, en particulier lorsque le système modélisé n'est encore pas en exploitation.

Notons enfin l'inadéquation totale des modèles analytiques pour décrire de façon détaillée une structure, ou pour représenter le fonctionnement d'un module de système, ou les interactions de différents modules, au niveau fin.

Un autre type de modèle théorique, dont l'audience s'est largement développée ces dernières années, permet de rendre compte de ces problèmes : ce sont les modèles basés sur les graphes de données et de contrôle.

I.2 LES MODELES DE GRAPHES DE CONTROLE

Les travaux de Pétri, qui a établi un formalisme très général au début des années soixante [C24], sont à la base des modèles graphiques de machines et de programmes. Ce formalisme, communément appelé "réseaux de Pétri", a été particularisé pour rendre compte plus aisément des caractéristiques des schémas de programmes parallèles, du contrôle d'exécution des co-routines dans un système d'exploitation, ou de l'activation parallèle d'éléments matériels d'un ordinateur [C2], [C10], [C14], [C17].

Un réseau de Pétri est un triplet
(T, P, A) où

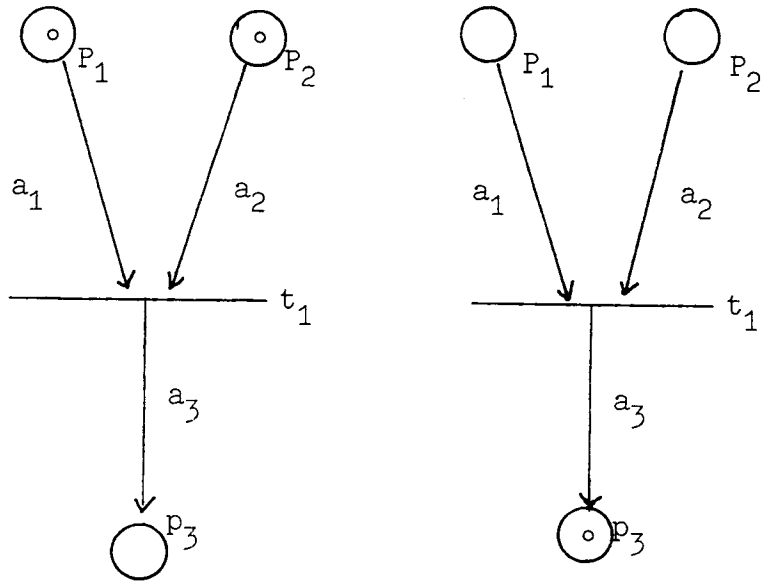
$T = \{t_1, t_2, \ldots t_n\}$ est un ensemble fini de transitions

$P = \{p_1, p_2, \ldots p_m\}$ est un ensemble fini de places

$A = \{a_1, {}_2, \ldots a_p\}$ est un ensemble fini d'arcs orientés qui joignent des éléments de T à des éléments de P et réciproquement.

Les places représentent des conditions et les transitions des événements. Les places peuvent contenir des jetons, qui sont capables de se déplacer dans le réseau en suivant les arcs orientés. Lorsque toutes les places en entrée d'une transition sont occupées par un jeton, la transition peut "tirer", c'est-à-dire qu'un jeton est ôté de chaque place d'entrée, et un jeton est mis dans chaque place de sortie de la transition. Il peut y avoir dans cette action perte ou création de jetons.

Exemple



La simulation d'un réseau de Pétri consiste à :

a/ définir un état initial, c'est-à-dire une première répartition de jetons dans des places ;

b/ à partir de l'état courant, définir le nouvel état en faisant tirer toutes les transitions qui peuvent le faire ;

c/ s'arrêter si aucune transition ne peut tirer, ou si le nombre maximal de changements d'état fixé à l'avance est atteint ; sinon revenir en b.

Un tel modèle, par sa généralité, permet des études théoriques sur des conditions de déterminisme, de conservation de jetons, et de sécurité, qui ont particulièrement été développés sur des sous-ensembles des réseaux de Pétri [C5]. Cependant, de même qu'au modèle de schéma de programme parallèle [C17], aucune interprétation ne lui est attachée, c'est-à-dire qu'aucune signification particulière n'est associée aux transitions, et que le contenu des éléments de mémorisation n'est pas spécifié. Cette généralité est en fait un inconvénient pour le concepteur, car le modèle n'est pas suffisamment proche de ses préoccupations.

Un modèle équivalent aux réseaux de Pétri, le graphe de UCLA [C10] donne une signification aux transitions, chaque opération étant assimilée à un ET ou à un OU exclusif.

Le système LOGOS [C14], [C23], équivalent au graphe de UCLA, utilise pour la construction des graphes des noeuds primitifs différents (ET, OU, SI ... ALORS ... SINON, APPEL de procédure, RETOUR de procédure, etc...). Ce dernier type de modèle possède un pouvoir évocateur bien supérieur aux précédents.

Des langages, inspirés par ces modèles graphiques, ont vu le jour ces dernières années. La philosophie de ces langages repose sur la séparation, dans une description de système, de la partie opérative et de la partie contrôle. La partie opérative est généralement exprimée dans un sous-ensemble d'un langage classique (APL, PL/1), la partie contrôle étant écrite dans un formalisme particulier, inspiré par les réseaux de Pétri [A27], ou par le système LOGOS [A30]. Les communications entre les deux parties se font par l'utilisation de mêmes noms de variables ou de procédures. A notre connaissance, seul le langage OSM [A22], dont la partie opérative est écrite sous une forme proche des langages de description de matériel (dont nous parlerons plus loin), et qui est plus particulièrement orienté vers la description des machines micro-programmées, possède à ce jour un simulateur opérationnel, lequel a été mis en oeuvre sur un ordinateur polonais.

I.3 LES MODELES DESCRIPTIFS

Par opposition aux modèles théoriques et aux modèles de simulation, un modèle descriptif n'a pas pour but essentiel de fournir une vérification de fonctionnement ou une évaluation de performance. Ecrit dans un langage formalisé, il permet de définir de façon précise une structure, et sert de moyen de communication non ambigüe entre personnes ou équipes travaillant sur un projet commun.

Les primitives du langage utilisé doivent permettre de donner les spécifications des composants, telles qu'on les trouve dans les catalogues des constructeurs, ainsi que les relations imposées entre ces composants. Ce langage sert aussi à créer, modifier, ou retrouver des données, dans une base de données de descriptions d'ordinateurs. Il est intéressant, de plus, qu'existent des mécanismes pour vérifier automatiquement qu'un modèle de structure est cohérent par rapport à la description de ses composants.

Knudsen [A20] estime que l'élaboration d'un modèle descriptif concerne essentiellement deux types d'utilisateurs. Pour le constructeur de nouveau matériel, un tel modèle doit servir de "croquis intelligent". Pour l'analyste ou pour le programmeur système qui travaille avec les composants décrits sous une telle forme, la compréhension d'une structure devrait être simplifiée, parce que plus clairement exprimée.

Dans [A7], Bell et Newell exposent des structures de systèmes extrêmement divers (HP 9100A, Burroughs B 6500, DEC PDP 8, CDC 6600 etc...) à l'aide de leur double langage de description PMS - ISP.
Fait intéressant, aucun de ces deux langages n'était compilé à l'époque.

I.4 LES MODELES DE SIMULATION

La simulation sur ordinateur est une technique largement répandue, et dont l'utilisation dépasse le cadre de l'informatique. Malcom Jones [D12] donne de la simulation la définition suivante :

> "C'est l'action de concevoir, construire, tester, valider, faire fonctionner et analyser un modèle formel du système étudié, qui a pour fonction de représenter seulement les aspects du système qui sont significatifs par rapport aux objectifs de l'étude".

Cette définition très large inclut un certain nombre d'opérations distinctes, qui inter-agissent lors du processus global de simulation, faisant intervenir de plus d'autres activités telles que le prélévement et l'analyse de données d'entrée [F6], l'analyse, par exemple statistique, des résultats [D11], et la validation du modèle [D14].

Pour ce qui nous concerne, c'est-à-dire la simulation des systèmes informatiques, le modèle formel sera toujours un programme exécutable sur ordinateur. De plus, nous nous limiterons à la simulation discrète, dite aussi simulation par événements, par opposition à la simulation continue (laquelle faisant intervenir la résolution d'équations aux différences finies, tend à approcher la solution d'équations différentielles de fonctions continues).

Un événement est une action qui se produit à un moment bien déterminé, et qui est supposée de durée nulle. La représentation d'un phénomène réel, qui lui prend un certain temps, sera appelée activité, et sera constituée par l'intervalle entre l'événement "début du phénomène" et l'événement "fin du phénomène". L'état du système change uniquement sur événement, donc de manière discrète [D14]. Dans la suite, nous ne parlerons que de simulation par événements, à laquelle nous nous référons par le seul mot simulation.

Quand est-il opportun d'analyser un système informatique par simulation ? Francis F. Martin [D16] met en garde ses lecteurs contre l'abus de son utilisation, alors que pour certains problèmes d'autres techniques seraient plus efficaces ou moins coûteuses. Certains critères sont cependant reconnus comme justifiant le choix d'une approche par simulation :

- Il est impossible de faire des essais sur le système réel parce que
  - . il n'existe pas encore
  - . cela perturberait son fonctionnement
  - . le coût d'une telle expérience est trop élevé ;

- aucune technique analytique ne permet de donner une solution satisfaisante ;

- l'élaboration d'un modèle de simulation, considérée comme une expérience pédagogique, permet de donner une meilleure compréhension du système, que l'on peut ainsi manipuler très facilement.

Nous pouvons classer les simulations de systèmes informatiques en quatre catégories principales, en fonction de leurs buts et des méthodes employées :

1/ Simulation du matériel et des micro-programmes en "pas à pas" : sa fonction essentielle est de vérifier le bon fonctionnement logique de la machine.

2/ Emulation : c'est la simulation, sur une machine, du macro-code d'une autre machine. Cette technique est employée dans le cas d'une gamme de machine (exemple : IBM 360) : toutes les micro-machines des ordinateurs de la gamme sont différentes, mais le code d'instructions qu'elles interprètent répond aux spécifications d'un même manuel de référence.
On trouve aussi des cas d'émulation logicielle, qui permettent aux constructeurs d'écrire le logiciel d'une machine avant qu'elle ne soit en état de fonctionnement [F8].

3/ Simulation en vue d'une évaluation de performance. En concurrence avec l'expérimentation directe et la modélisation mathématique, elle est extrêmement controversée. La méthode la plus répandue consiste, à partir de traces de programmes, ou sur des nombres générés selon une loi de probabilité donnée, à faire des relevés statistiques sur l'état du système, et notamment à mesurer les taux d'occupation des différentes ressources, la longueur des files d'attente qui leur sont associées et les temps d'attente des processus.

4/ Simulation des pannes.
Cette rubrique regroupe des simulations d'ordres très divers. On peut par exemple, à partir de données provenant de tests de composants, vouloir déterminer l'intervalle de temps pendant lequel le système fonctionnera correctement [D8]. D'autres simulations ont pour but de faire apparaître les conséquences du mauvais fonctionnement d'un circuit, ou d'un module programmé, par comparaison avec une unité sans panne [F20], [F23].

II . LES OUTILS D'AIDE A LA CONCEPTION

Parmi les différents types de modèles que nous venons de citer, les modèles de description et de simulation sont les outils du concepteur d'un nouveau système. Historiquement, ces modèles étaient réalisés en langage machine ou en assembleur ; puis, avec l'apparition des langages compilés, Fortran en particulier, l'écriture se faisait dans un langage général de programmation. Le langage APL, qui offre un éventail très riche d'opérateurs, a plus tard été utilisé [F24].
Cependant, certains mécanismes spécifiques de la simulation, notamment la gestion du temps simulé et du parallélisme des activités, devaient être programmés par l'utilisateur, ce qui rendait ces modèles difficiles à mettre au point et à modifier. C'est pourquoi, dès le début des années soixante, la nécessité d'une plus grande souplesse dans l'élaboration des modèles a motivé la création de langages mieux adaptés.

La séparation traditionnelle entre les équipes de conception du matériel et du logiciel a entraîné le développement de deux familles de langages très distinctes, qui se caractérisent par le degré d'abstraction du modèle par rapport à la machine décrite.

II.1 LES MODELES ABSTRAITS

Les modèles les plus abstraits servent à simuler les ressources d'un système (mémoire, CPU, disques) et les algorithmes, le plus souvent en vue d'une prédiction de performance. La description de ces ressources et de ces algorithmes est une description de comportement, leur réalisation matérielle n'étant pas signifiante. Les exemples d'application de simulations fonctionnelles sont nombreux : distribution du flot de travaux entre deux processeurs couplés de puissance inégale, de façon à minimiser les temps d'inactivité du processeur le plus rapide [D17] ; évaluation de la gamme IBM 360 [D19] ; évaluation des effets d'une modification de répartition de charge dans un réseau [D5] ; détermination du goulot d'étranglement dans un système multi-utilisateurs en temps partagé [D20] etc...

Quoique certains auteurs préférent écrire de tels modèles de simulation en Fortran [D18], pour des raisons d'occupation en mémoire et de rapidité d'exécution, malgré l'effort supplémentaire de programmation que cela implique, les utilisateurs de langages généraux de simulation sont de plus en plus nombreux. Les langages les plus connus sont, GPSS excepté [B7], [B8], [B9], [B10], une extension d'un langage général de programmation : SOL [B13], [B14] et SIMULA [B1], [B2], [B3] , [B5], [B6] , [B17] sont une extension d'Algol, SIMSCRIPT [B12], [B13], [B14], [B15], GASP [B11] et CSL [B4] partent de Fortran, DESPL/1 [B19] et SIMPL/1 [B20] sont une extension de PL/1.

Ces langages ont un certain nombre de caractéristiques communes :

- Le temps simulé est incrémenté de valeurs variables.
- Le mécanisme de gestion du temps est automatisé. Les événements ont leur réalisation (qui peut être conditionnelle) prévue dans le futur, et sont rangés dans un échéancier. Une routine du simulateur est chargée de prélever dans l'échéancier le prochain événement ; le temps simulé est avancé à sa date de réalisation, l'événement est retiré de l'échéancier et les calculs correspondants sont effectués. Lorsque toutes les actions relatives à cet événement sont achevées, l'échéancier est à nouveau inspecté. En cas de simultanéité de plusieurs événements, certains langages offrent des primitives qui permettent de leur affecter une priorité.

- Les objets simulés présents à un instant donné sont soit actifs, soit dormants, soit mis en file d'attente (attente sur une ressource occupée ou absente). L'utilisateur dispose de primitives pour activer ou désactiver un objet, le mettre dans une file d'attente ou l'en retirer, créer ou détruire un objet etc...

- Selon le langage, une bibliothèque de sous-programmes plus ou moins riche permet de générer des nombres aléatoires selon les fonctions de densité de probabilité les plus usuelles, de récolter automatiquement des données statistiques, d'aider à la mise au point des programmes.

- Le mode de prise en compte des événements implique que des processus parallèles se déroulant en même temps sont simulés en quasi-parallèlisme : on en fait exécuter d'abord un puis un autre. En particulier, l'ordre d'exécution peut avoir une influence sur le résultat final de la simulation.

Le domaine d'application de ces langages généraux dépasse le cadre des systèmes informatiques. Il existe cependant des langages de même niveau d'abstraction spécialisés pour ce domaine. Citons parmi les plus connus CSS, développé par IBM à partir des concepts de GPSS, très orienté vers la simulation des séries 360/370 [D19], et ECSS, une extension de SIMSCRIPT II développée par la RAND Corporation [B16], plus souple et plus général.

II.2 LES MODELES DE DESCRIPTION DU MATERIEL

A l'opposé des modèles abstraits les modèles de description du matériel spécifient la structure physique de la machine. Le niveau de détail peut se situer de la porte logique au composant complexe. C'est d'ailleurs ce type de modèle qui le premier a reçu une grande attention, à une époque où le coût d'un ordinateur dépendait essentiellement du coût du matériel. Dans ce domaine peut être plus encore que pour le logiciel, des outils de description bien adaptés sont devenus indispensables ; et les langages spécialisés ont rapidement supplanté les langages d'assemblage ou Fortran pour l'écriture des modèles.

L'aspect documentaire de ces langages est extrêmement important. On peut en effet associer au compilateur des programmes de traitement et de visualisation, pour dessiner les modules et les fils de communication entre ces modules, tâche qui, faite à la main, est excessivement longue et fastidieuse.

De plus lorsqu'à un langage de description du matériel est associé un simulateur, celui-ci devient un outil d'expérimentation qui évite la construction d'une maquette. Différentes solutions peuvent être testées pour leur faisabilité technologique et leur efficacité. La simulation permet aussi de vérifier le bon fonctionnement logique de la description, et la mise au point des micro-programmes.

Soulignons enfin le très grand intérêt de ces langages pour l'enseignement de la structure et des fonctions des machines.

Les langages de description du matériel possédent, pour la plupart, un certain nombre de caractéristiques qui les distinguent des autres langages de simulation.

- Les objets simulés sont les bascules, les registres, les mémoires, les horloges, les fils de connexion, les bus, les organes de retard.

- Quoique certains langages autorisent les variables arithmétiques et réelles [A19], [A35], la plupart de ceux pour lesquels existe un compilateur ne traitent que des booléens scalaires et tableaux. Les opérateurs primitifs sont donc les opérateurs : OU, ET, NEGATION, OU EXCLUSIF. Quelques opérateurs de transformation, tels que le décalage, le décalage circulaire et la concaténation sont aussi couramment employés.

- Il est toujours fait une différence entre les notions de transfert et de connexion. Un transfert est le chargement d'une valeur dans un élément de mémorisation, à un instant bien déterminé. La valeur chargée est conservée jusqu'au prochain transfert, c'est une valeur permanente. Une connexion est le passage d'une valeur sur des fils de connexion. Cette valeur n'est pas conservée, c'est une valeur transitoire.

- Le parallélisme des actions élémentaires est une particularité importante que ces langages doivent exprimer. On distingue à ce propos, selon la terminologie de Barbacci [E1] les langages procéduraux, pour lesquels l'ordre d'écriture des instructions influe sur le résultat : ISP [A6], APDL [E5], SFD - Algol [A28], des langages non procéduraux, pour lesquels l'ordre lexicographique des instructions est indifférent : AHPL [A18], CASSANDRE [A1], [A25], [A26], CDL [A10], [A11], [A12], DDL [A14], [A15], ERES [A5].

- Le séquencement d'actions ou de groupes d'actions élémentaires est introduit par la notion d'automate dans les langages non procéduraux. A chaque état de l'automate correspond un ensemble d'instructions étiqueté par un nom d'état ou une condition logique. Pour les langages procéduraux, le séquencement est réalisé par le passage d'un bloc temporel au suivant (sauf dans le cas d'une instruction de contrôle de déroutement).

- La description structurelle, correspondant à l'organisation physique de la machine, tient une large place dans un modèle de description de matériel. La déclaration des objets simulés et la spécification de leurs interconnexions est toujours prévue dans un langage spécialisé pour l'écriture de ce type de modèle. La description fonctionnelle des algorithmes réalisés s'exprime par les relations entre les composants, sous l'action d'instructions de contrôle et d'impulsions de synchronisation (parfois de manière assez lourde).

II.3 NECESSITE D'UN SYSTEME INTEGRE D'AIDE A LA CONCEPTION

Les deux classes de langages que nous venons de présenter, correspondant à des préoccupations et des domaines d'application très différenciés, s'accordent mal à un processus global de conception d'un système informatique. Or, à l'heure actuelle, le coût croissant du logiciel, qui doit répondre à des applications de plus en plus complexes et diversifiées, et l'avènement d'une technologie hautement intégrée, qui à la fois réduit le prix des composants matériels et permet au concepteur de ne plus redescendre jusqu'au niveau de la porte logique, rendent indispensable un rapprochement des spécialistes du matériel et du logiciel.

Un système informatique doit, dès le départ, être pensé dans son intégralité, et de façon aussi modulaire que possible, afin que soit choisie, pour chaque fonction à réaliser, la mise en oeuvre (cablée, micro-programmée ou programmée) la mieux adaptée pour satisfaire aux contraintes de coût, de fiabilité et de performance.

Sans vouloir entrer dans la mauvaise querelle opposant les partisans d'une méthode ascendante de conception aux tenants d'une méthode descendante, il nous semble incontestable qu'au cours de l'élaboration du système, les modules entrant dans sa composition passeront par différents stades de définition. Il est donc nécessaire de pouvoir disposer d'un langage, ou d'un ensemble cohérent de langages, permettant l'écriture de modèles à des niveaux de détail très divers. C'est dans ce but que nous avons défini le langage LASCAR.

Un "bon" langage d'aide à la conception des systèmes informatiques doit, selon nous, posséder les propriétés suivantes :

1/ Il doit permettre différents niveaux de description : matériel, micro-programmes, fonctions, algorithmes (l'ensemble PMS - ISP est très intéressant de ce point de vue, APDL possède partiellement cette propriété).

2/ Il doit être suffisamment descriptif pour qu'au niveau le plus détaillé, à l'aide de programmes de traitement spécifiques, la réalisation physique et les connexions puissent être visualisées et au moins partiellement automatisées. C'est le cas de AHPL, CASSANDRE [A9], LOGAL [A21].

3/ Il doit être modulaire. Il faut pouvoir décrire, tester et simuler les composants séparément, afin que la modification d'un composant, ou son remplacement par un composant fonctionnellement équivalent, mais de structure interne différente, ne remette pas en cause le modèle tout entier. Les langages SIMULA, SIMSCRIPT, GPSS, CASSANDRE, ERES, DDL, PMS, ISP sont modulaires, contrairement à CDL, AHPC, APDL.

4/ Lorsque l'on décrit un composant à des niveaux différents, il faut pouvoir aisément valider le modèle le plus abstrait à partir du modèle le plus détaillé. Nous ne connaissons aucun langage satisfaisant à cette condition.

5/ La gestion du parallélisme et des événements simultanés doit être automatique et leur expression aisée. C'est le cas de tous les langages de description de matériel.

6/ Le langage doit être doté d'un simulateur suffisamment souple pour que la bonne conception logique du modèle puisse être vérifiée en pas à pas (CDL, CASSANDRE, APDL) et suffisamment efficace pour que les durées de fonctionnement significatives puissent être évaluées (les langages abstraits répondent mieux à cette contrainte).

7/ Il est souhaitable enfin que l'utilisateur dispose du simulateur à la fois sous un système interactif pour mettre au point, tester ou modifier rapidement le modèle, et sous un système de traitement par lots, pour lancer de longues simulations. A notre connaissance, seul CASSANDRE a ces deux versions du même simulateur.

Parmi tous les langages que nous avons évoqués, aucun ne répond à tous ces critères. Notre but étant d'offrir au concepteur un outil bien adapté, la solution élégante eut été de définir un langage totalement nouveau. Notre manque d'expérience en la matière, et les faibles moyens dont nous disposions nous ont cependant ramenés à des vues moins ambitieuses. Nous avons donc décidé d'adapter à nos exigences un langage existant, de façon à pouvoir disposer, dans des délais raisonnables, d'un outil en état de fonctionnement. CASSANDRE a été choisi pour les deux raisons suivantes :

a - Il possède déjà les propriétés 2, 3, 5, 7.

b - Nous disposions de tous les modules source du compilateur et du simulateur, que nous pouvions modifier et réutiliser comme base d'un nouvel ensemble de programmes de traitement.

Nous exposerons dans la première partie de cette thèse le langage LASCAR, en justifiant les choix qui ont été fait par les objectifs auxquels nous essayions de répondre.
Dans la deuxième partie, nous détaillerons une application de LASCAR à la modélisation d'un système multiprocesseur, et donnerons les résultats de simulation que nous avons obtenus.
La troisième partie sera consacrée à l'analyse de mesures qui ont été prélevées sur le simulateur, et à l'évaluation critique de ce simulateur.
Nous conclurons enfin sur une appréciation du travail présenté et sur ce qu'il reste encore à faire pour obtenir un ensemble complet et cohérent d'outils d'aide à la conception des systèmes informatiques.

PREMIERE PARTIE

DÉFINITION DU LANGAGE LASCAR

INTRODUCTION

Certains principes orientés par les propriétés dont nous cherchions à doter LASCAR nous ont guidés dans la définition de ce langage :

- Nous devions obtenir un simulateur aussi efficace que possible en moins de deux ans. Nous avons donc choisi d'étendre la version synchrone de CASSANDRE qui était, lorsque ce travail a été entrepris, la seule version opérationnelle, en nous donnant le moyen d'exprimer de façon simple la durée prise par des opérations complexes.

- Toujours dans un but de performance à la simulation, nous avons introduit la notion de séquentialité, car le caractère non procédural de CASSANDRE conduisait parfois à des descriptions inutilement lourdes.

- Les nouvelles primitives que nous avons définies introduisent dans une description un niveau d'abstraction supérieur. Aussi, nous avons, autant que faire se pouvait, distingué dans la syntaxe toutes les notions qui permettent de ne plus représenter la réalisation matérielle d'un algorithme, afin que l'utilisateur soit toujours conscient des concepts qu'il manipule.

- LASCAR étant une extension de CASSANDRE, il était important que toute description écrite dans ce dernier langage soit acceptée telle quelle par le nouveau simulateur, afin d'éviter les erreurs de transcription et de conserver intactes les propriétés intéressantes de CASSANDRE de pouvoir descriptif et de représentation du parallélisme au niveau fin. Nous ne nous sommes donc permis aucune modification de la syntaxe de CASSANDRE.

- De manière à faciliter la validation d'un modèle abstrait par rapport à un modèle détaillé, nous avons gardé inchangée l'interface des unités de description.

- Enfin, chaque fois qu'il fallait choisir entre élégance et gain de temps à la simulation, nous avons préféré le second critère. D'où par exemple la distinction entre les types entier et tableau entier, qui nous évite de construire un descripteur pour les scalaires.

Nous ne rappellerons du langage CASSANDRE que les notions qui sont nécessaires à la compréhension de LASCAR, et renvoyons le lecteur aux ouvrages dans lesquels CASSANDRE a été exposé en détails [A25], [A26]. Après quoi nous présenterons de manière informelle les primitives de LASCAR, l'important pour nous étant surtout de souligner les concepts nouveaux, et l'utilisation qui peut en être faite à la simulation. Une définition rigoureuse de la syntaxe de LASCAR pourra être trouvée dans [F7], rapport n° 3, Chapitre III. La grammaire du langage est donnée en annexe I.

I . RAPPELS SUR LE LANGAGE CASSANDRE

Avertissement

Certaines notions, par exemples les termes "signal" et "état", sont employées dans le langage CASSANDRE avec une signification différente de leur acception courante. Nous en rappellerons brièvement la sémantique, au sens du langage CASSANDRE, toutes justifications utiles pouvant être trouvées dans [A25] et [A26].

Le domaine d'application du langage CASSANDRE est celui de la modélisation des "systèmes logiques". Nous extrayons de [A26, introduction] la définition suivante :

> "Un système logique peut être défini entièrement à l'aide de l'algèbre de Boole et de la théorie des automates finis (...) Si on réalise physiquement le système logique décrit, on aboutit à une machine digitale. Mais dans de nombreux cas, ceux par exemple où l'on peut en rester à une étude de réseau, la conception ne débouche pas forcément sur une réalisation physique ; c'est en ce sens que la notion de système logique est plus générale que celle de machine digitale".

I.1 NOTION D'UNITE

Tout programme CASSANDRE est une unité. Une unité est la description d'un système logique, ou d'un sous-ensemble de système logique, pouvant être considéré comme un tout physiquement ou fonctionnellement.

Toute description peut être découpée en morceaux de taille et de complexité arbitraires. Et à tout découpage correspond une structure arborescente d'unités imbriquées. Ce choix de structure arborescente reflète la décomposition réelle d'un circuit complexe en ses constituants plus élémentaires.

Une unité se décompose en trois parties, dont seule la première est obligatoire :

- L'entête d'unité :
  L'entête d'unité déclare le nom de l'unité par un identificateur. Cet identificateur est global à tout le programme. Le nom de l'unité est suivi de la liste, entre parenthèses, de ses entrées et de ses sorties, avec leurs dimensions. Les identificateurs qui repèrent les entrées et les sorties d'une unité sont locaux à cette unité.
- Une liste de déclarations et de spécifications
- Une liste d'instructions.

I.2 DECLARATIONS ET SPECIFICATIONS

Les variables manipulées dans le langage CASSANDRE se répartissent en cinq types :

a) Les horloges

L'évolution d'un système logique décrit en CASSANDRE est exprimée de manière discrète. Le cycle d'horloge fixe l'intervalle de temps élémentaire. Les instants, supposés infiniment courts, où une variable de type horloge a pour valeur 1 (tops d'horloge), sont les instants où l'état du système logique est susceptible d'être modifié. On suppose que l'intervalle de temps séparant deux tops d'horloge est suffisant pour que le système logique ait le temps de se stabiliser.

b) Les signaux

Les variables de type signal sont des niveaux logiques qui restent établis pendant les intervalles de temps séparant deux tops d'horloges. Elles correspondent aux signaux physiques portés par les éléments de connexion : fils, circuit combinatoires. Leur valeur est supposée toujours disponible aux instants de synchronisation.

c) Les registres

Ce sont des éléments de mémorisation : bascules, registres, mémoires. Les variables de type registre peuvent garder indéfiniment la valeur qu'elles mémorisent, et cette valeur ne peut être modifiée que sous l'action d'une impulsion (top d'horloge).

Les variables de type horloge, signal et registre ont, à un instant donné, la valeur d'une constante booléenne de mêmes dimensions.

d) Les états

Dans un système logique séquentiel, les opérations élémentaires qui s'exécutent simultanément sont regroupées pour former des instructions composées. L'ordre d'exécution établi entre ces instructions composées définit le séquencement du système logique. Ce séquencement est exprimé à l'aide d'un automate d'état fini ; un état de l'automate est associé à chaque instruction composée. Les variables de type état permettent de repérer les états de l'automate, indépendamment de tout codage. Elles sont considérées comme des éléments de mémorisation. Leur valeur est une étiquette, ou un ensemble ordonné d'étiquettes pour les variables à plusieurs dimensions, nommant une instruction composée.

e) Les sous-unités

Ce sont des morceaux de description considérés de l'extérieur comme des "boites noires" et dont on ne connait que le nom et les entrées-sorties.

SPECIFICATIONS

Par défaut, les entrées et les sorties d'une unité sont des signaux. On autorise cependant que certaines entrées-sorties soient des horloges. Si l'on veut exprimer qu'une (ou plusieurs) entrée-sortie est une horloge, on doit la spécifier horlogemère.

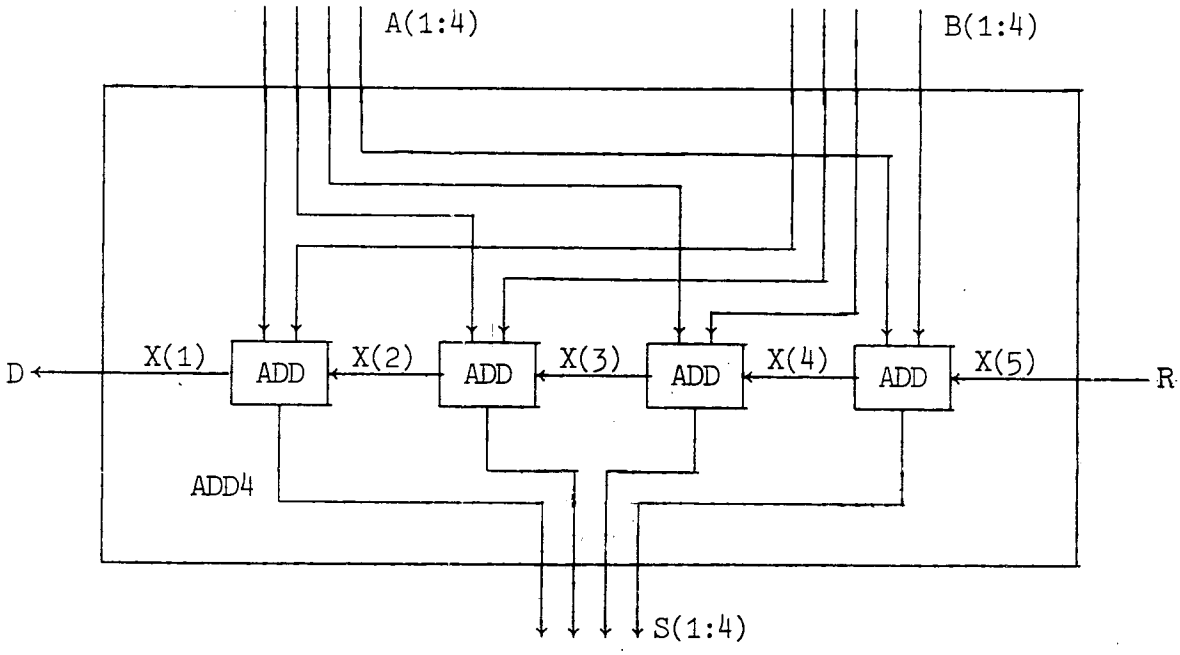
DECLARATIONS

Les déclarations indiquent le type et les dimensions des variables utilisées dans une description. Les noms des variables de type registre, signal, horloge et état sont locaux à une unité. Par contre, les sous-unités, déclarées externe, peuvent avoir été décrites et compilées séparément, et leur nom est global à tout le programme.

EXEMPLE

Le petit programme ci-dessous décrit un additionneur parallèle 4 bits réalisé à l'aide de 4 additionneurs complets de 1 bit. Les entrées A(1:4) et B(1:4) sont les deux vecteurs de 4 bits à additionner. L'entrée R est la retenue éventuelle qui provient du résultat d'une addition qui aurait été effectuée en amont. Les sorties de l'additionneur sont le vecteur de bits résultat S(1:4) et le bit de débordement D.

Les quatre additionneurs de 1 bit sont connectés à un bit de A, un bit de B et un bit de S, de même rang. Le report de la retenue d'un additionneur au suivant est effectué à l'aide d'un signal interne X(1:5), dont le premier élément est connecté à D et le dernier à R.



```
unité   ADD4 (A(1:4), B(1:4), R ; S(1:4), D) ;

signal   X(1:5) ; "signal interne de connexion"
"déclaration de la sous-unité ADD, avec 3 signaux d'entrée et 2 signaux
de sortie, supposée définie ailleurs"

externe   ADD (, , ; , ) ;
"connexion des 4 exemplaires de l'additionneur 1 bit"

pourtout   I   de  1  a  4

début   ADD |I| (A(I), B(I), X(I+1) ; S(I), X(I)) ;

fin ;
D := X(1) ; R := X(5) ;
```

I.3 LES INSTRUCTIONS

Les instructions d'une unité décrivent la structure des circuits combinatoires construits à partir du matériel défini dans les déclarations, et les fonctions réalisées par ce matériel.

Les instructions appartiennent à l'une des deux catégories suivantes :

- Les instructions de branchement.
  Elles correspondent à des connexions de signaux, et d'horloges internes branchées sur l'horlogemère. Elles équivalent à des équations booléennes.

- Les instructions d'affectation.
  Ce sont les instructions conditionnées par une horloge. Elles correspondent au chargement des éléments de mémorisation et aux créations d'impulsions d'horloges internes en phase avec l'horlogemère.

Une deuxième dichotomie sépare les instructions d'une unité en deux classes distinctes :

- Les instructions toujours vraies.
  Ces instructions, qui apparaissent toujours immédiatement après les déclarations, sont effectuées à tous les instants de la simulation.

- Les instructions sous état.
  Une unité peut posséder plusieurs états internes, chaque état débutant par une étiquette. Cette unité possède alors une variable interne de type état, qui est prédéfinie (et qui n'a donc pas besoin d'être déclarée). Cette variable, à tout instant, a pour valeur l'une, et une seule, des étiquettes qui apparaissent dans l'unité.

  Une instruction sous état est effectuée si et seulement si la variable d'état interne de l'unité a pour valeur l'étiquette sous la portée de laquelle cette instruction est placée. Le passage d'un état à l'autre valide donc des groupes d'instructions différents : l'unité est décrite comme un automate séquentiel.

A chaque cycle, les instructions toujours vraies et les instructions qui se trouvent sous la portée de l'étiquette qui nomme l'état courant sont exécutées de manière collatérale, et le résultat est indépendant de l'ordre lexicographique des instructions.

a) Opérateurs et expressions

Définition

Deux variables sont compatibles (ou ont des dimensions compatibles) si

1/ Elles ont le même nombre de dimensions.
2/ Leurs dimensions de même rang possèdent le même nombre d'éléments.

Dans CASSANDRE, comme dans APL, tous les opérateurs peuvent porter sur des scalaires aussi bien que sur des tableaux. Lorsque les opérateurs dyadiques portent sur des variables non scalaires, celles-ci doivent avoir des dimensions compatibles, et les opérations sont effectuées entre éléments de même rang (seule exception : la concaténation).

Opérateurs dyadiques booléens

Les opérateurs dyadiques booléens ont la signification suivante :

. et
+ ou
≠ ou exclusif
= égal
> supérieur
< inférieur
≥ supérieur ou égal
≤ inférieur ou égal

Opérateurs monadiques booléens

- négation
  Appliquée à un tableau, la négation porte sur chacun des éléments de ce tableau

/α réduction par rapport à α, α étant un opérateur dyadique booléen.
  Cet opérateur est défini pour les variables non scalaires, et il s'applique à la première direction d'un tableau de dimensions quelconques.
  Le résultat de la réduction d'un tableau à p dimensions est un tableau à (p - 1) dimensions.

Opérateurs monadiques de transformation

Soient i et j deux entiers sans signe, k une constante arithmétique.

Les opérateurs monadiques de transformation sont définis, pour des variables non scalaires, de la façon suivante :

* D ↑ k ↑ décalage, selon la première dimension, de k positions à gauche si k est positif, à droite si k est négatif.
  La première dimension du tableau est diminuée de k éléments.

* R ↑ k ↑ décalage circulaire, selon la première dimension, de k éléments à gauche si k est positif, à droite si k est négatif. La première dimension du tableau garde le même nombre d'éléments.

* P ↑ i ↑ ↑ j ↑ permutation des dimensions i et j d'un tableau à p dimensions, tel que p ≥ i et p ≥ j.

Concaténation

L'opérateur de concaténation ε permet de concaténer selon la première dimension des variables ayant le même nombre de dimensions, et telles que toutes les dimensions, sauf la première, soient compatibles. Un scalaire est pour cet opérateur assimilé à un vecteur de un élément.

Expressions booléennes simples

Une expression booléenne simple est une combinaison de variables booléennes, de constantes booléennes et d'opérateurs. Les variables et les constantes doivent vérifier les règles de compatibilité qui ont été précédemment énoncées.

La priorité des opérateurs est donnée par le tableau ci-dessous (les chiffres croissants indiquent une plus forte priorité) :

PRIORITE	1	2	3	4	5
OPERATEUR	+	.	< ≥ ≤ = > ≠	ε	- /α * P * R * D

Expressions logiques d'état

Une expression logique d'état provient de la comparaison de variables d'état et d'étiquettes. Une telle expression permet de tester la valeur d'une variable d'état de l'unité courante ou bien la valeur du registre d'état prédéfini d'une sous-unité.

Expressions conditionnelles

En CASSANDRE, l'opérateur si admet comme condition une expression booléenne simple, ou une expression logique d'état [A25].
L'expression qui sert de condition doit être scalaire. L'opérateur si permet d'écrire des expressions conditionnelles booléennes et des expressions conditionnelles d'état : le résultat du calcul de la condition valide l'un des termes de l'alternative (partie alors et partie sinon). Les deux termes doivent être des expressions de même type et de dimensions compatibles. La partie sinon n'est obligatoire que pour les expressions conditionnelles d'état.

b) Les branchements simples

Les instructions de branchement définissent des connexions de fils. Ce sont des instructions indépendantes de toute impulsion de synchronisation. On distingue trois types de branchements :

Connexion de signal

La connexion de signal correspond à la soudure d'un fil, ou d'une nappe de fils, sur la sortie d'un circuit combinatoire. L'instruction de connexion est en fait une équation booléenne. La connexion est désignée par le symbole :=. La variable en partie gauche de ce symbole et l'expression en partie droite doivent avoir des dimensions compatibles.

Connexion d'horloge

La connexion d'horloge est analogue à la connexion de signal. Mais la partie droite du symbole := doit être une expression d'horloge, c'est-à-dire une expression booléenne dans laquelle les seules variables qui peuvent apparaître sont les variables de type horloge.

Connexion d'unité

La connexion d'une unité déclarée externe est en fait une connexion multiple de l'ensemble de ses entrées-sorties. La connexion est effectuée en remplaçant, dans la déclaration d'externe, les entrées par des expressions booléennes ou des expressions d'horloge de l'unité englobante, et les sorties par des variables de type signal ou de type horloge de l'unité englobante. Pour chaque entrée-sortie, les règles de compatibilité de type et de dimensions doivent être respectées.

Si une sous-unité est utilisée en plusieurs exemplaires, on attribue à chacun d'eux un numéro, appelé numéro de duplication, pour pouvoir les référencer individuellement. Le numéro de duplication est connu statiquement ; c'est donc une constante entière, ou une expression arithmétique construite uniquement à l'aide de constantes entières et de variables de contrôle de boucles pourtout, appelée expression d'itération.

Dans une connexion d'unité, on peut ne pas associer d'identificateur à l'une des entrées ou des sorties, en la désignant par le symbole *. L'occurence du symbole * est équivalente au signal (ou à l'horloge) qu'il remplace, et il est considéré comme une variable de type signal (ou une variable de type horloge) à laquelle n'est associé aucun identificateur. On l'appelle, parfois de façon impropre, "signal-unité".

Exemple

```
unité   EX (H(1:3), E(1:16), A, B ; S(1:3)) ;
horlogemère   H ;
signal   C(1:8), D(1:2), X, Y ;
externe   EXT (horloge, , ; (1:2)) ;
horloge   K ;
EXT ↑ 1 ↑ (H(1), E(1), A ; D) ;
EXT ↑ 2 ↑ (K, X, * ; S(2:3)) := si Y alors B sinon A ;
K := H(2) + H(3) ;
X := /. E ;
```

c) Les affectations simples

Les instructions d'affectation servent à charger les variables de type registre et état. La valeur est mémorisée sous l'action d'une impulsion. Ces instructions servent aussi à créer une impulsion d'horloge interne en phase avec une autre impulsion. Une instruction d'affectation est toujours effectuée en présence d'une condition horloge.

Une affectation particulière est l'ordre allera qui, dans une unité contenant des états, sert à charger le registre d'état implicite avec une valeur d'étiquette. L'ordre allera, considéré comme un chargement de registre, sert à introduire un séquencement dans la validation des différents états.

Exemple

```
horlogemère   H ;
horloge   K ;
signal   A, B, C(1:20, 0:3) ;
registre   S(1:20, 1:8), T(1:20, 1:4), U, V ;
état   E(1:3) ;
ET1  :  <H> S(, 1:4) <= T.C,
            allera ET2 ;
ET2  :  <H> K := A + B, allera E(3) ;
        <K> V <= 1 ;
ET3  :  ..........
```

d) L'instruction conditionnelle

L'instruction conditionnelle fait dépendre l'exécution de un ou plusieurs groupes d'instructions de la valeur d'un vecteur de conditions logiques, qui peut être réduit à un scalaire.

A chaque élément de ce vecteur est associée une liste d'instructions entre parenthèses dans la partie alors, et une liste d'instructions entre parenthèses dans la partie sinon. La valeur de chacun des éléments du vecteur de conditions logiques conditionne l'exécution de l'une ou l'autre liste d'instructions qui lui sont associées. Une liste d'instructions peut être vide ; dans ce cas, seule la paire de parenthèses apparait. La partie sinon n'est pas obligatoire.

La règle de compatibilité est la suivante : la partie alors et éventuellement la partie sinon, doit comporter autant de listes d'instructions délimitées par des parenthèses que le vecteur de conditions comporte d'éléments.

Exemple

```
signal   COND(1:3), A, B, C(1:4), D(1:4, 1:5), E, F, G ;
externe  HADD(, ; ,) ; registre R(1:24, 1:4), S(1:24, 0:7) ;
horlogemère  H ;
si  COND  alors (A := B ; C := D(, 1))  (1)
                (HADD(A, E ; F, G))     (2)
                (D(, 5) := 1100)        (3)
          sinon (A := -B)               (4)
                ( )                     (5)
                (<H>  R <= S(, 4:7)) ;  (6)
```

Si COND contient par exemple 110, les instructions des lignes (1), (2) et (6) seront exécutées. Si COND contient 101, seules les instructions des lignes (1) et (3) seront exécutées, la ligne (5) étant une instruction vide.

e) L'instruction FAIRE

L'instruction faire est spécifique de la partie automate d'une unité. Cette instruction permet, à partir d'un état donné, et sans sortir de cet état, d'effectuer toutes les instructions d'un autre état. C'est l'analogue d'un appel de sous-programme.

L'instruction faire ne modifie pas la valeur du registre d'état prédéfini de l'unité. Elle ne se trouve donc pas sous la portée d'une condition horloge.

Exemple

```
horlogemère  H ;
signal  REP, S ; registre  AD(1:24), INST(0:31) ;
E1  :  si REP alors faire EK ;
E2  :  ......
 .
 .
 .
EK  :  S := 1 ;
       <H> AD <= 0000 0000 0000 ε INST(20:31),
           allera E2 ;
```

Une écriture équivalente serait :

```
E1  :  si REP alors
       (S := 1 ;
       <H> AD <= 0000 0000 0000 ε INST(20:31),
       allera E2) ;
```

E2 .

f) L'instruction d'itération

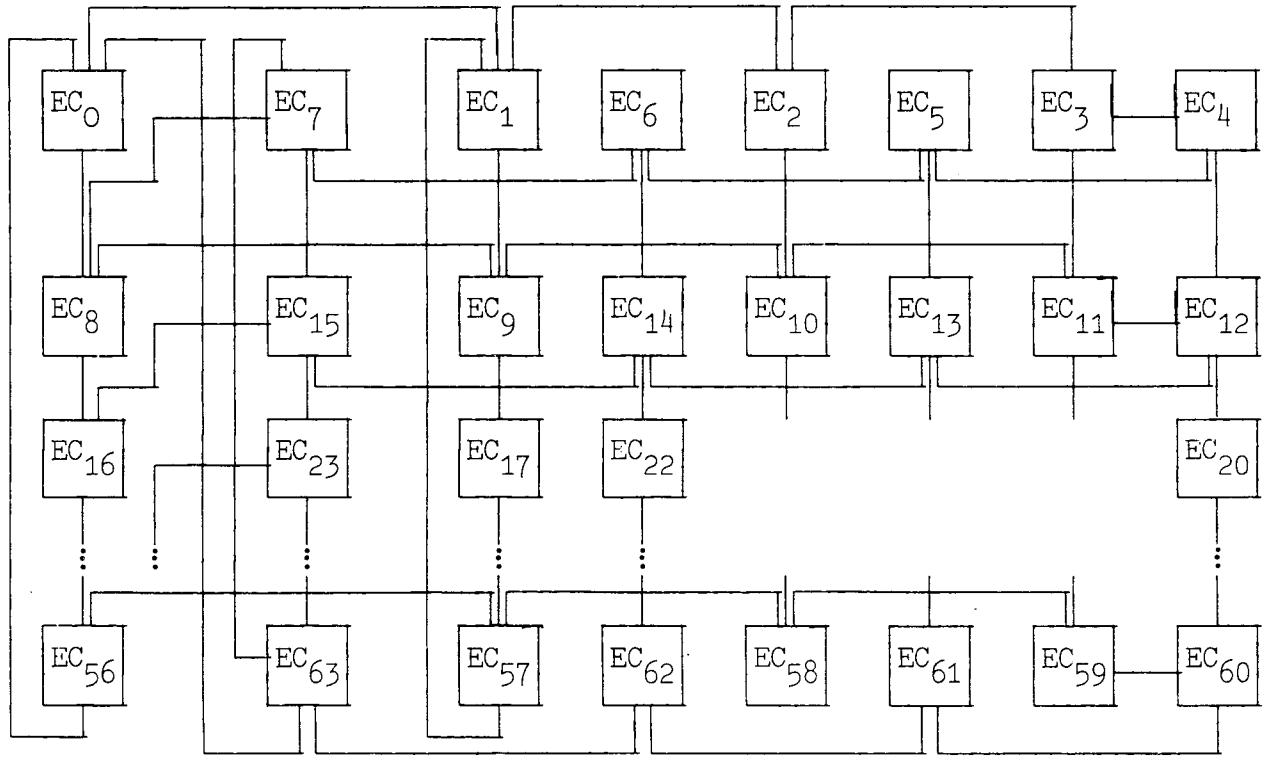
L'instruction d'itération, telle qu'elle est utilisée dans CASSANDRE, sert à décrire de manière condensée les propriétés géométriques répétitives d'un système (connexion avec décalages des numéros de fils, utilisation de plusieurs exemplaires d'une unité etc...). Elle est introduite par le mot clé pourtout.
La variable de contrôle d'une instruction d'itération n'est pas déclarée. Elle n'a d'existence que pour le bloc d'instructions qui se trouve sous la portée de l'instruction d'itération, délimité par les mots clé début et fin, et l'ordre dans lequel elle prend ses différentes valeurs est indifférent.

La valeur initiale, l'incrément et la limite de l'index d'itération doivent être des valeurs entières positives, connues statiquement ; ce sont des expressions arithmétiques très particulières, composées uniquement de constantes arithmétiques positives, d'index d'itération d'instructions itératives englobantes, et d'opérateurs arithmétiques.

Il est possible de décrire des exceptions dans une structure répétitive, en testant une condition sur un ou plusieurs index d'itération, introduite par le mot clé sia.

Exemple

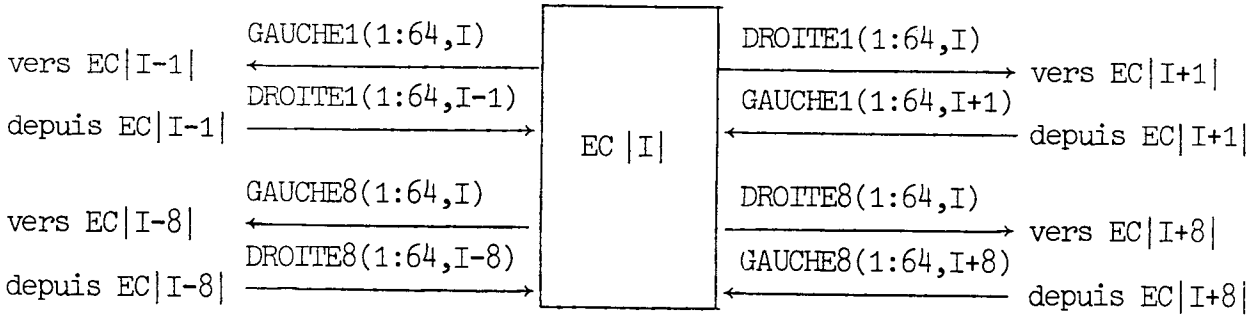
La description de la page suivante est inspirée de la structure de la machine ILLIAC4 [A7, pp 320-329]. Cet ordinateur est composé de 256 éléments de calcul EC, organisés en 4 tableaux de 64 éléments. Nous nous limiterons à un tableau, dont nous ne décrirons pas l'unité de contrôle, mais uniquement les connexions. Chaque élément de calcul EC du tableau est relié, par un chemin de données de 64 bits, à 4 de ses voisins. Le processeur numéro i est connecté aux processeurs numéros $i+1$, $i-1$, $i+8$, $i-8$. Aux extrémités, les liaisons sont circulaires, de telle sorte que le processeur numéro 63 est connecté aux processeurs numéros 0, 62, 7 et 55. Les chemins de données sont bi-directionnels.



Sur ce schéma, chaque trait représente un ensemble de 64 fils de connexion.

Dans la description, en CASSANDRE, de cette structure, les connexions sont dédoublées, car les variables de type signal sont uni-directionnelles.

L'ensemble des fils de connexions est organisé en 4 nappes de 64 fois 64 bits, le deuxième indice étant associé au numéro de l'élément de calcul d'où partent les fils.




```
'EXTERNE' EC((0:63),(0:63),(0:63),(0:63);
              (0:63),(0:63),(0:63),(0:63));
'SIGNAL' GAUCHE1(1:64,0:63),GAUCHE8(1:64,0:63), DROITE1(1:64,0:63),
         DROITE8(1:64,0:63);
"CONNEXION DES EXTREMES"
EC|0|(DROITE1(,63),     "VENANT DE EC|63| "
      GAUCHE1(,1),      "VENANT DE EC|1| "
      DROITE8(,56),     "VENANT DE EC|56| "
      GAUCHE8(,8);      "VENANT DE EC|8| "
      GAUCHE1(,0),      "VERS EC|63| "
      DROITE1(,0),      "VERS EC|1| "
      GAUCHE8(,0),      "VERS EC|56| "
     DROITE8(,0) );     "VERS EC|8| "
EC|63|(DROITE1(,62),     "VENANT DE EC|62| "
       GAUCHE1(,0),      "VENANT DE EC|0| "
       DROITE8(,55),     "VENANT DE EC|55| "
       GAUCHE8(,7);      "VENANT DE EC|7| "
       GAUCHE1(,63),     "VERS EC|62| "
       DROITE1(,63),      "VERS EC|0| "
       GAUCHE8(,63),      "VERS EC|55| "
       DROITE8(,63) );     "VERS EC|7| "
"CONNEXION DE TOUS LES ELEMENTS DE CALCUL SAUF LES DEUX
  EXTREMES"
'POURTOUT' I 'DE' 1 'A' 62
'DEBUT'
     EC|I| (DROITE1(,I-1),
            GAUCHE1(,I+1),
            DROITE8(, 'SIA' I<8 'ALORS' 56+I 'SINON' I-8),
            GAUCHE8(, (I+8) 'REM' 64);
            GAUCHE1(,I),
            DROITE1(,I),
            GAUCHE8(,I),
            DROITE8(,I));
'FIN';
```

II . STRUCTURE D'UN PROGRAMME LASCAR

II.1 INTERFACE DE L'UNITE

La modularité du langage CASSANDRE est l'une de ses qualités les plus interessantes. Nous avons donc tenu à la conserver dans LASCAR. Un programme LASCAR est donc lui aussi une unité, qui peut être découpée en sous-unités, en nombre et en niveaux d'imbrication arbitraires.

Le premier choix que nous avons dû faire concerne l'entête des unités. Nous pouvions définir de nouveaux types de variables, par exemple des entiers, comme entrées-sorties d'une unité. Et de même que nous avions la spécification horlogemère, nous aurions pu ajouter des spécifications supplémentaires.

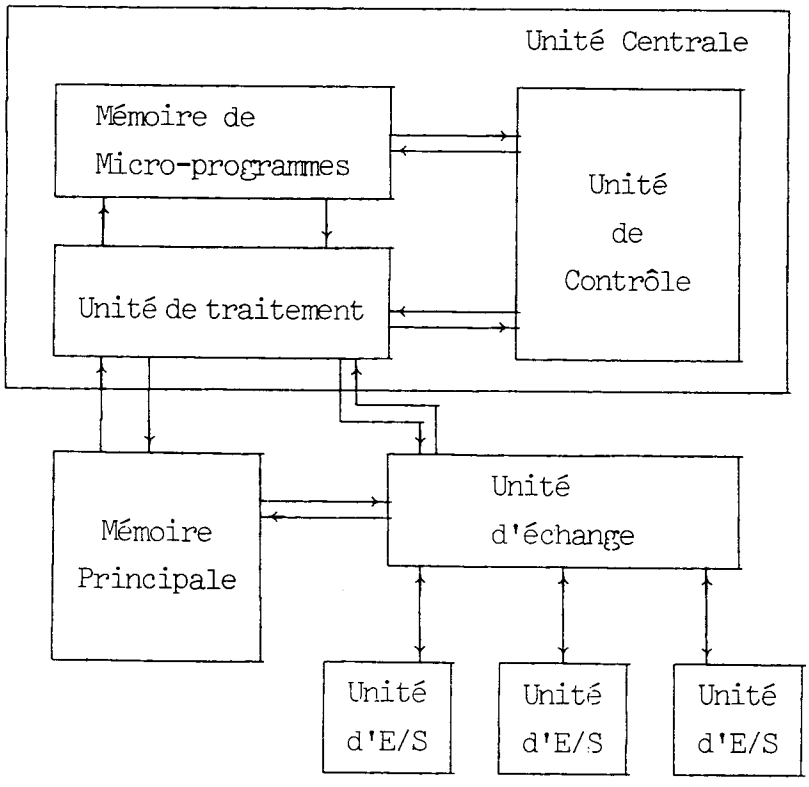
Nous avons cependant préféré conserver inchangée l'interface des unités, pour deux raisons :

1/ Connexion d'unités écrites en CASSANDRE et en LASCAR

Au cours de l'élaboration d'un projet, les différents constituants d'un ordinateur ne sont pas simultanément définis avec le même degré de précision. Ainsi, LASCAR peut servir de langage de spécification, et CASSANDRE est utilisé pour vérifier la bonne conception logique dans le détail. Il peut donc être utile de faire coexister dans un même modèle des unités décrites en CASSANDRE et des unités décrites en LASCAR. Dans ce cas, les entrées et les sorties des unités doivent être de types acceptables par les deux langages.

Exemple

Soit le schéma de machine très général suivant :



On peut imaginer qu'à un stade donné, seules l'unité de traitement et l'unité de contrôle aient été conçues et vérifiées, et soient décrites en CASSANDRE, et que toutes les autres unités soient en cours de spécification, et décrites en LASCAR.

2/ Equivalence de deux unités

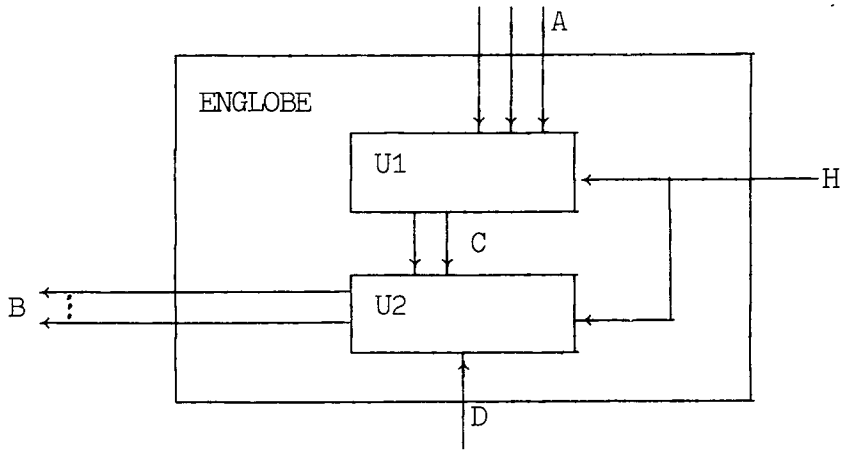
Dans un modèle d'ordinateur, il est important que l'utilisateur puisse facilement remplacer une unité écrite en LASCAR par une unité équivalente écrite en CASSANDRE, et vice-versa. Or ceci n'est envisageable que si les deux unités, vues de l'extérieur, sont interchangeables sans qu'aucune modification doive être apportée au reste du modèle. De manière plus précise, nous adopterons la définition suivante :

Deux unités sont équivalentes si :
- elles ont les mêmes entrées-sorties
- à tout instant, pour des valeurs identiques de leurs entrées, leurs sorties ont la même valeur.

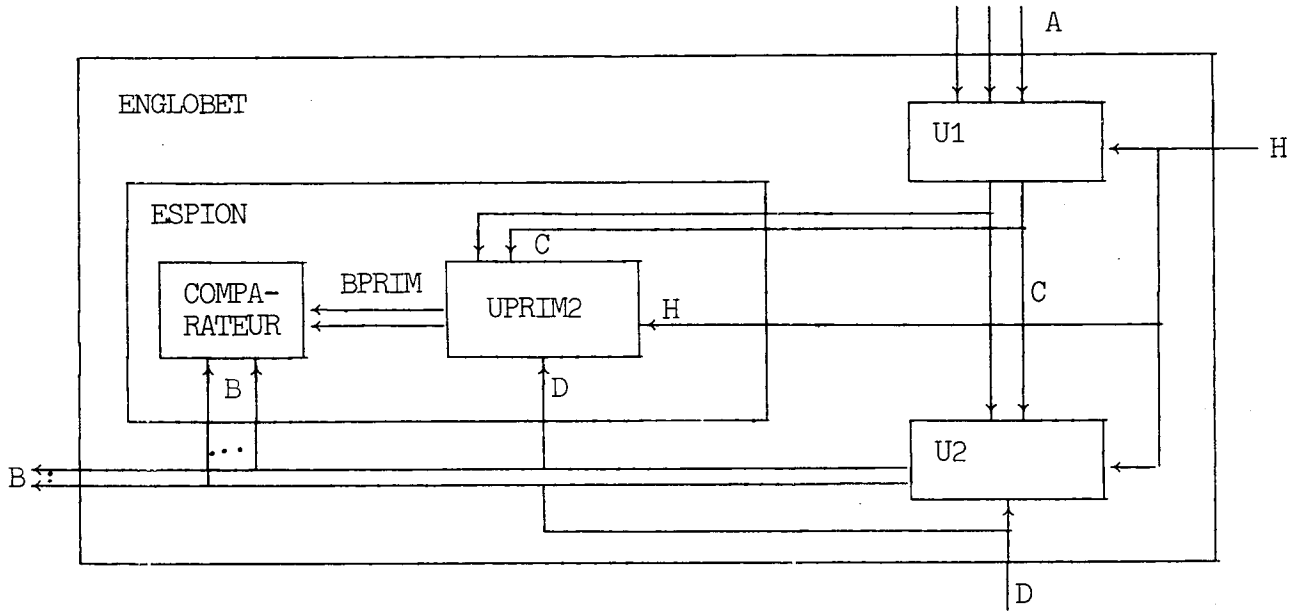
Une telle équivalence n'est pas décidable dans le cas général. Mais, pour une application particulière, il est possible de vérifier partiellement l'équivalence de deux unités sur un échantillon représentatif de valeurs des entrées qui ont une signification pour le modèle considéré. Nous ne saurions trop recommander d'effectuer cette vérification, dont nous donnons ci-dessous, sur un exemple simple, une méthode possible.

Exemple

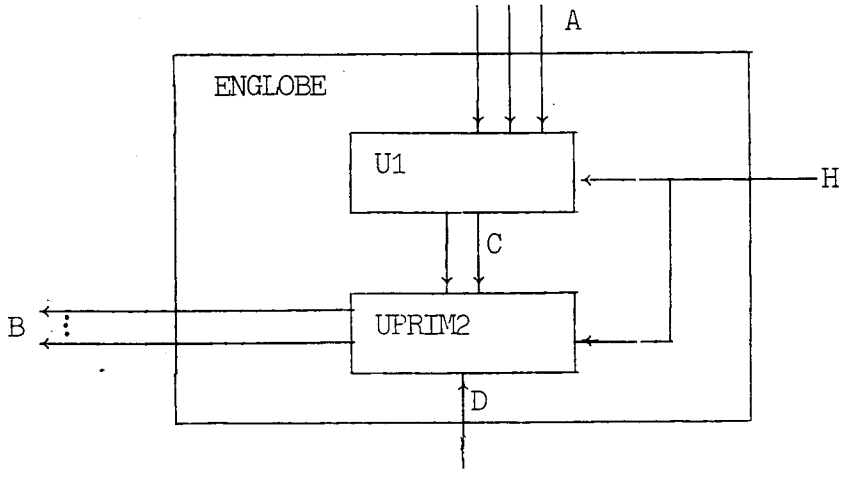
Dans la figure de la page 20, l'unité ENGLOBE contient les deux unités U1 et U2. Les fils A, B, C, D sont des entrées, des sorties et des connexions. H est l'horloge de synchronisation commune. Supposons que nous ayons une unité UPRIM2 que nous voulons substituer à U2. Nous rajoutons à ENGLOBE une boite fictive, qui doit être considérée comme ne faisant pas partie du modèle, mais plutôt comme un mécanisme d'espionnage. Cette boite ESPION reçoit les entrées et les sorties de U2, mais n'inter-agit pas sur le modèle. L'unité ESPION contient l'unité UPRIM2, qui reçoit en entrée les mêmes entrées que U2, et un comparateur qui vérifie à tout instant l'identité des valeurs des sorties de U2 et de UPRIM2.



Modèle Primitif



Mise en oeuvre de la vérification d'équivalence



Modèle équivalent

```
unité   ENGLOBET (H, A(1:3), D ; B(1:12)) ;

horlogemère   H ;

signal   C(1:2) ;

externe   U1 (horloge, (1:3) ; (1:2)),
          U2 (horloge, (1:2), ; (1:12)),
          ESPION (horloge, (1:2),, (1:12) ; ) ;

"connexions"

U1(H, A ; C) ; U2(H, C, D, B) ;
ESPION(H, C, D, B ; ) ;


unité   ESPION(H, C(1:2), D, B(1:12) ; X) ;

horlogemère   H ;  compteur CYCLE ;

signal   BPRIM(1:12) ;

externe   UPRIM2 (horloge, (1:2), ; (1:12)) ;

entier   X, Y ; procédure ERREUR(3) ;

"connexion"

UPRIM2(H, C, D ; BPRIM) ;

"vérification de l'équivalence"

<H> p1 (CYCLE), X $ = $(B), Y $ = $(BPRIM),
    sie X ≠ Y alors appel ERREUR (CYCLE, X, Y) ;
```

II.2 ORGANISATION DE L'UNITE

La structure d'une unité écrite en LASCAR est identique à celle d'une unité écrite en CASSANDRE. Nous la rappelons brièvement :

- l'entête d'unité, qui ne diffère pas de l'entête d'une unité écrite en CASSANDRE,
- une liste de déclarations et de spécifications, qui en plus des déclarations de variables CASSANDRE peut contenir des déclarations de compteurs, d'entiers et de procédures,
- une liste d'instructions.

III . LES OBJETS DU LANGAGE LASCAR

Tous les objets du langage CASSANDRE sont traités par LASCAR. On pourra donc trouver dans un programme LASCAR des déclarations de signaux, d'horloges, de registres, d'états et de sous-unités. A ces notions, nous avons jugé utile de rajouter des notions plus proches de celles des langages généraux de programmation, pour écrire des descriptions de type algorithmique.

III.1 LES ENTIERS

Un entier est une variable qui contient à un instant donné une valeur entière positive ou négative.

Le but, en utilisant des entiers, est de décrire de manière fonctionnelle une unité, indépendamment de sa réalisation matérielle. Le concept de variable immédiate en sortie d'un circuit combinatoire est dans ce cas superflu, à l'intérieur d'une unité écrite en LASCAR. Un entier a donc été défini comme un élément de mémorisation, et sa valeur ne peut changer qu'à un top d'horloge. Un entier se comporte donc de ce point de vue comme un registre.

L'interface d'une unité écrite en LASCAR étant composé de signaux et d'horloges, il est nécessaire d'établir une règle de correspondance entre entier et vecteur booléen codant cet entier. A cet égard, deux solutions, répondant à des objectifs contradictoires, s'offraient à nous :

- La solution la plus générale consiste à laisser à l'utilisateur le choix du nombre de bits codant un entier, et le choix du codage des nombres négatifs. Ainsi, la déclaration d'un entier se fait sous une forme qui peut être :

entier (18, complément à 1) A ;

Cette solution implique la simulation de toutes les opérations arithmétiques d'un modèle, comme cela est fait en CASSANDRE pour les opérations logiques sur les registres et les signaux. Cette généralité se paye donc par une importante inefficacité à la simulation.

• La solution la plus performante consiste à fixer comme nombre de bits codant un entier le nombre de bits du mot mémoire de l'ordinateur sur lequel est mis en oeuvre le langage, et à adopter pour le codage des nombres négatifs la même convention. Ainsi, pour la réalisation qui a été faite du compilateur de LASCAR sur les systèmes IBM 360/370, un entier est codé sur 32 bits, en complément à deux pour les nombres négatifs. Pour déclarer un entier, il suffit d'écrire :

entier A ;

Cette solution est beaucoup moins élégante, et oblige l'utilisateur à prendre des précautions lorsqu'il fait une conversion d'entier à vecteur booléen. Par contre, elle permet une utilisation directe du macro-code de la machine hôte, d'où un gain de temps considérable à la simulation.

Nous avons déjà dit, dans l'introduction de ce chapitre, que nous avions pris comme option de préférer le critère de performance au critère d'élégance. Nous avons donc choisi la deuxième solution.

Dans la suite de ce mémoire, nous considérerons toujours qu'un entier est réalisé par un mot de 32 bits, codé en complément à 2, ce qui correspond à la mise en oeuvre que nous avons faite sur les systèmes IBM 360/370. Il est bien évident que le codage d'un entier pourrait être différent sur une autre machine.

III.2 LES TABLEAUX ENTIERS

Un tableau entier est une suite ordonnée d'entiers. Le nombre de dimensions, et l'étendue des dimensions d'un tableau entier ne sont pas limités. Un tableau entier est considéré comme un élément de mémorisation, et sa valeur ne peut changer qu'en présence d'une impulsion.

Contrairement aux variables de CASSANDRE, pour lesquelles un même mot clé déclare les variables scalaires et tableaux, les types entier et tableau entier sont distingués dans LASCAR, rejoignant en cela ce qui est coutumier dans les langages généraux de programmation. En effet, cette séparation évite la construction d'un descripteur pour les entiers scalaires, ce qui implique à la fois un gain en occupation mémoire et en temps de simulation.

III.3 LES COMPTEURS

Un compteur est un élément de mémorisation très particulier dont la valeur est un entier. Les seules opérations autorisées sur un compteur sont l'initialisation, l'incrémentation de 1 et la décrémentation de 1, en présence d'un top d'horloge.

Les variables de type compteur jouent deux rôles essentiels dans une description :

- placées en des points choisis, elles servent de compteur d'événements et espionnent le fonctionnement du modèle. Leur présence dans une description est comparable à celle d'un "hardware monitor" dans un système réel.
- utilisées pour comptabiliser les tops d'horloge, elles servent à introduire des retards dans une unité décrite de manière entièrement algorithmique.

III.4 LES PROCEDURES

De même que nous avons jugé utile d'introduire un type particulier de variable, le type compteur, il nous a semblé tout à fait indispensable de pouvoir disposer, dans le langage LASCAR, de la notion de procédure. Les procédures facilitent les initialisations des unités d'un modèle, et sa mise au point par l'impression de traces d'états et de valeurs de variables. Mais surtout, les procédures sont le seul moyen économique de prélever dans le modèle des relevés statistiques et de les imprimer sous des conditions (ou à des instants) prévus à l'avance. De plus, les procédures sont utilisées pour simuler très rapidement des unités dont on ne veut pas décrire la réalisation physique, et qui n'interviennent dans un modèle que par la fonction qu'elles remplissent.

Une procédure n'est pas un sous-programme composé d'instructions écrites en LASCAR, mais un sous-programme écrit dans un langage général de programmation, et compilé séparément. Pour le moment, les seuls langages acceptés sont FORTRAN, et l'Assembleur 360. Cette liste n'est pas limitative, et des interfaces avec d'autres langages peuvent être rajoutés facilement si cela s'avère utile.

Pour des raisons de structure de données, les seuls paramètres acceptables pour une procédure FORTRAN sont les compteurs, les entiers, les tableaux entiers et les constantes entières. Par contre, toute variable est un paramètre acceptable pour une procédure écrite en assembleur.

Pour que le passage par l'interface convenable soit assuré, un mot clé spécifique sert à déclarer le langage dans lequel est écrite la procédure : le mot clé fortran annonce des procédures écrites en FORTRAN, le mot clé procédure annonce des procédures écrites en Assembleur. Le nom d'une procédure est global au programme LASCAR, et doit être celui sous lequel la procédure a séparément été compilée. De plus, il faut indiquer lors de la déclaration le nombre de paramètres de la procédure, s'il y en a.

Exemple 1

Supposons que l'on veuille simuler le comportement d'un processeur sur une trace d'instructions. Pour une trace contenant un grand nombre d'instructions, il peut être très coûteux en place, si ce n'est impossible, de décrire la mémoire principale du calculateur simulé. Dans ce cas, il est préférable de simuler la recherche d'une instruction par un appel de procédure, qui ira chercher sur un fichier les caractéristiques de la prochaine instruction à exécuter : adresse de l'instruction, code opération, adresse des deux opérandes par exemple. On déclarera alors :

procédure prochain (4) ;

Exemple 2

Des prises de mesure lors de la simulation d'une unité ont permis de remplir le tableau qui représente l'histogramme des temps séparant deux arrivées successives du même événement. Admettons que l'on veuille imprimer cet histogramme, lorsqu'une condition particulière est vérifiée. Le sous-programme FORTRAN suivant permet d'effectuer ce travail de façon extrêmement simple. Les paramètres I1 et I2 représentent respectivement le tableau à imprimer et la dimension de ce tableau.

```
      SUBROUTINE  HISTO (I1, I2)
      DIMENSION  I1(1)
1     FORMAT  (25 H HISTOGRAMME DES ARRIVEES/)
2     FORMAT  (8 (I8) )
      PRINT  1
      PRINT  2  (I1(J), J = 1, I2)
      RETURN
      END
```

Ce sous programme, qui aura été compilé séparément par un compilateur FORTRAN, est déclaré dans l'unité LASCAR qui l'utilise :

fortran histo (2) ;

IV . OPERATEURS ET EXPRESSIONS

Tous les opérateurs et toutes les expressions de CASSANDRE ont été conservés dans LASCAR. Nous ne parlerons donc ici que de ce qui est nouveau dans ce langage, ou de l'utilisation nouvelle qui en est faite.

IV.1 OPERATEURS PORTANT SUR LES VARIABLES ARITHMETIQUES

a) Opérateurs de comptage

plusun (en abbrégé p1) : incrémentation de 1
moinsun (en abbrégé m1) : décrémentation de 1

Ces deux opérateurs ne peuvent s'appliquer qu'aux variables de type compteur.

Le rôle de ces variables, qui servent à espionner un modèle ou à introduire des retards, étant très spécifique, leur utilisation dans une description ne doit pas pénaliser le temps de simulation. C'est pourquoi nous avons introduit ces deux opérateurs, qui sont réalisés de façon très performante.

b) Autres opérateurs monadiques

Les deux autres opérateurs monadiques s'appliquent à des variables de type entier et tableau entier. Dans le cas d'un tableau entier, l'opération est effectuée sur tous les éléments du tableau, et le résultat est un tableau entier de mêmes dimensions.

\- opposé
abs valeur absolue

c) Opérateurs dyadiques arithmétiques

Ces opérateurs portent sur des variables de type compteur, entier ou tableau entier. Dans le dernier cas, les opérations sont effectuées entre les éléments de même rang, deux à deux, et les deux opérandes doivent avoir des dimensions compatibles.

Le résultat est un entier, ou un tableau entier de mêmes dimensions que les deux opérandes.

+	addition
-	soustraction
.	multiplication
./	division entière
rem	reste de la division entière

d) Opérateurs dyadiques de comparaison

Ces opérateurs portent sur des variables de type compteur, entier ou tableau entier, sous les mêmes conditions que pour les opérateurs dyadiques arithmétiques. Le résultat de l'opération est une constante booléenne de mêmes dimensions que les deux opérandes.

<	inférieur
≤	inférieur ou égal
=	égal
≥	supérieur ou égal
>	supérieur
≠	différent

IV.2 OPERATEURS DE CONVERSION

a) Le "valnum" : $

L'opérateur $ (appelé valnum) est un opérateur monadique applicable à une variable (ou expression) booléenne vectorielle ou tableau. Le résultat est l'entier, ou le tableau d'entiers, codé en binaire par cette variable. L'opérateur valnum effectue donc une conversion du type booléen au type entier. Au cours de cette conversion, il y a perte d'une dimension : un vecteur booléen est converti en un entier scalaire, un tableau booléen à deux dimensions est converti en un entier vectoriel, etc...

Un entier étant codé sur 32 bits, la règle de conversion est la suivante :

- si on applique $ à un booléen dont la première dimension est supérieure à 32, on tronque les bits de poids fort de la première dimension, de façon à ne garder que les 32 bits de faible poids.

- si on applique $ à un booléen dont la première dimension est inférieure à 32, on étend cette première dimension jusqu'à 32 en complétant les forts poids manquants par des zéros.

L'utilisateur a donc la charge, s'il décrit une machine dont le mot mémoire est inférieur à 32 bits, de concaténer éventullement à gauche un vecteur dont tous les bits sont à 1, s'il veut obtenir un nombre négatif. Si au contraire son mot mémoire est supérieur à 32 bits, il lui faudra utiliser 2 ou plusieurs entiers, dans le cas où la troncature des bits de poids forts aurait des conséquences fâcheuses pour la validité des résultats de simulation.

Exemple

Signal A(1:12), B(1:8, 2:4), C(1:38) ;

Si A(1:12) = 0 0 0 1 0 0 0 0 1 1 0 1
$(A) = $2^0 + 2^2 + 2^3 + 2^8 = 269$

Si B est le tableau :
1 0 0 1 0 0 1 1
0 0 0 0 1 1 0 0
0 1 0 0 0 1 1 1
$(B) est le vecteur : (147 12 71)

Si C = 0011 0111 1111 1111 1011 1111 0011 1111 0100 10
$(C) est calculé par :

. troncature à 32 bits, on obtient :
11 1111 1111 1011 1111 0011 1111 0100 10

. le premier bit est à un : le nombre est négatif.
On prend le complément à 2 :
00 0000 0000 0100 0000 1100 0000 10 11 10
$(C) = - 1 060 910

b) Le "valbin" : !

L'opérateur ! (appelé valbin) est un opérateur monadique applicable à une variable (ou expression) entière scalaire ou tableau. Cet opérateur est l'inverse du valnum. Il donne pour résultat le vecteur, ou le tableau booléen, qui code en binaire l'entier auquel il est appliqué en rajoutant une première dimension : un entier scalaire est converti en un vecteur booléen, un vecteur entier est converti en un tableau booléen à deux dimensions, etc...

Les règles de conversion sont les suivantes :

- un entier est d'abord traduit en un vecteur booléen de 32 bits, codé en complément à 2 si l'entier est négatif, le premier bit étant un bit de signe.
- la première dimension du résultat de la conversion est déterminée par la première dimension d_1 de la variable booléenne à laquelle il est affecté.
  Si $d_1 < 32$, on tronque les $32-d_1$ bits de poids fort.
  Si $d_1 > 32$, on étend à gauche par d_1-32 zéros.

Pour l'utilisation de cette conversion, l'utilisateur doit prendre le même type de précautions, mais en sens inverse, que celles qui avaient été préconisées pour l'emploi du "valnum".

Il nous a été suggéré de paramètrer le "valbin" de façon à indiquer, dans l'opérateur, le nombre de bits du résultat de la conversion. Il nous semble cependant qu'un tel choix inciterait l'utilisateur à insérer abusivement des connexions d'entiers dans les expressions booléennes, et à mélanger les niveaux de description au sein d'une même unité en perdant la maitrise des concepts qu'il manipule. Nous avons au contraire cherché à distinguer au maximum entre description logique et description fonctionnelle.

Exemple

```
signal   A(1:36), B(1:6, 2:4)
entier   E ;
tableau entier   F(1:3) ;
Si  E = - 2  et  F = (8, 131, -1)
A := !(E) → A = 0000 11....11 10
                ⎧ 0 0 1 0 0 0 ⎫
B := !(F) → B = ⎨ 0 0 0 0 1 1 ⎬
                ⎩ 1 1 1 1 1 1 ⎭
```

IV.3 EXPRESSIONS

a) Expressions arithmétiques simples

Une expression arithmétique simple est une combinaison de variables arithmétiques, de constantes arithmétiques, d'opérateurs définis au paragraphe IV.1 et d'expressions booléennes simples après conversion. Les variables et les constantes doivent avoir des dimensions compatibles.

La priorité des opérateurs arithmétiques entrant dans la composition des expressions arithmétiques simples est donnée par le tableau ci-dessous :

PRIORITE	4	5	6
OPERATEUR	+	.	abs
	-	./	
		rem	

Exemple

```
entier   X, Y ;
tableau entier  A(0:7), TABLAD(0:3, 0:15) ;
registre   T(0:47, 0:3) ;
X - Y rem 4
(TABLAD(, 0) + $(T)) ./ A(0:3)
```

b) Expressions logiques

Une expression logique provient de la comparaison d'expressions simples arithmétiques de dimensions compatibles. Le résultat est une constante booléenne de mêmes dimensions.

Plusieurs expressions logiques peuvent être combinées par les opérateurs logiques et et ou pour former une nouvelle expression logique.

Dans ce contexte, il est nécessaire d'introduire des opérateurs et et ou, car les opérateurs . et + qui signifient "et" et "ou" lorsqu'ils sont appliqués à des variables booléennes ont déjà un autre sens dans les expressions arithmétiques.

La priorité des opérateurs est la suivante :

PRIORITE	1	2	3
OPERATEUR	ou	et	< > ≤ ≥ = ≠

Exemple

entier X, Y, Z ;

compteur V, W ;

tableau entier A(1:3), B(1:3), C(0:6) ;

V = 0

(X <Y ou X <Z) et W ≥ 3

A + B ≠ C(0:2)

c) Expressions conditionnelles

Nous avons défini un deuxième opérateur condition : sie.
Cet opérateur admet comme condition une expression logique (résultant de la comparaison de deux expressions arithmétiques). La condition sur laquelle porte l'opérateur sie étant construite à partir de variables de type compteur, entier et élément de tableau entier, l'emploi de cet opérateur correspond à une description pour laquelle aucune réalisation matérielle de mécanisme de comparaison n'est supposée. C'est pourquoi nous avons introduit, pour cet opérateur, un mot clé différent, permettant de le distinguer de l'opérateur si.

Si la condition a pour valeur vrai, l'expression qui suit le mot clé alors est validée ; dans le cas contraire, c'est l'expression qui suit le mot clé sinon. Ces deux expressions peuvent être des expressions booléennes, arithmétiques ou des expressions d'état. Elles doivent cependant être de même type, et de dimensions compatibles.

Nous avons de même étendu la sémantique de l'opérateur si de CASSANDRE, qui peut conditionner une alternative de deux expressions arithmétiques. Dans ce cas, l'opérateur si est assimilable à l'opérateur de même nom d'Algol 60, et la partie sinon est obligatoire.

Exemple

```
entier   X, Y, Z ;
tableau entier   A(1:3), B(1:3), C(0:6) ;
registre   R, S, T(1:4), U(1:4 , 1:2) ;
si R alors A sinon B
sie X < Z alors T sinon U(, 1)
```

V . LES INSTRUCTIONS

Les instructions d'un modèle écrit en LASCAR décrivent les fonctions réalisées par chacune des unités du modèle. Un certain degré de description structurelle est conservé : c'est le découpage du programme en unités, et l'écriture des instructions de connexion de ces unités. Par contre, à l'intérieur d'une unité écrite en LASCAR, toute description structurelle peut disparaître, pour laisser place à une description des algorithmes permettant d'obtenir les valeurs des signaux de sortie en fonction des valeurs des signaux d'entrée. De plus, l'emploi des opérateurs arithmétiques a souvent pour conséquence un découpage moins fin des unités en sous-unités.

Ainsi, en CASSANDRE, l'incrémentation d'un compteur fait intervenir non seulement l'élément de mémorisation dont la valeur est augmentée de un, mais tout le circuit combinatoire qui assure la propagation de la retenue d'une position binaire à la suivante dans cet élément de mémorisation ; en LASCAR, l'emploi d'un opérateur réalise cette incrémentation. Dans le même esprit, l'addition de deux entiers dispense l'utilisateur de décrire un additionneur, la multiplication de deux entiers remplace un multiplieur, etc... Un degré d'abstraction supérieur encore est obtenu grâce à l'emploi des procédures : nous en avons donné une illustration dans l'exemple 1 du paragraphe III.4. Une description en LASCAR peut donc être purement fonctionnelle.

Dans cet esprit, nous avons apporté, par rapport à CASSANDRE, une modification sémantique fondamentale : les instructions portant sur les variables de type compteur, entier et tableau entier qui sont valides à un instant donné sont exécutées séquentiellement à l'intérieur d'une même unité, sous l'action d'un top d'horloge.

En effet, dans CASSANDRE, le parallélisme interne à une unité correspond au parallélisme de la propagation des informations sur les chemins de donnés et dans les circuits combinatoires ; et la représentation d'un composant matériel impose par exemple qu'un même élément de mémorisation ne puisse, sous l'action d'une impulsion, recevoir qu'une seule valeur, faute de quoi le circuit logique dont on veut vérifier la bonne conception n'est pas déterministe. Cette contrainte implique que l'unité de temps de simulation soit l'intervalle de temps séparant deux impulsions consécutives de l'horloge de plus haute fréquence, ce qui peut conduire à des descriptions extrêmement lourdes et détaillées.

Si l'on cherche à s'abstraire de la réalisation matérielle d'une unité, et si l'on veut simuler celle-ci très rapidement, il faut pouvoir se dégager de ce type de contrainte. Une description fonctionnelle peut en effet nécessiter l'emploi de variables de travail qui n'ont aucune réalité physique, mais dont on veut disposer immédiatement de la valeur. L'unité de temps simulé qui a un sens pour ce type de description est le plus grand commun diviseur des temps séparant deux prises en compte successives des entrées de l'unité, et deux validations successives de ses sorties. En d'autres termes, les instants significatifs sont les instants où l'unité communique avec l'extérieur. Ces instants peuvent être beaucoup plus espacés que ceux qu'il aurait fallu considérer dans une description écrite en CASSANDRE. Il faut donc qu'une même variable mémorisée ait pu changer plusieurs fois de valeur, à l'intérieur d'une unité, entre les instants où cette unité se synchronise avec le monde extérieur. C'est la raison pour laquelle nous avons introduit la séquentialité des instructions purement LASCAR.

Nous n'avons toutefois pas modifié la sémantique des instructions de CASSANDRE, qui continuent de s'exécuter en parallèle. Et la collatéralité qui existe entre deux unités reste totale.

V.1 L'AFFECTATION D'ENTIER

L'affectation d'une variable de type compteur, entier ou tableau entier a la même signification que dans les langages généraux de programmation. Les affectations d'entiers sont des opérations séquentielles. Un entier peut, dans le même cycle d'horloge, recevoir plusieurs valeurs, et sa nouvelle valeur est disponible immédiatement. L'ordre d'écriture des affectations d'entiers est donc important.

De manière à bien distinguer l'affectation d'entier de l'affectation de registre, nous l'avons désignée par un nouveau symbole : $=. En partie gauche de ce symbole, on peut donc trouver une variable de type entier ou tableau entier, qui doit avoir des dimensions compatibles avec celles de l'expression arithmétique qui se trouve en partie droite.

L'incrémentation et la décrémentation de 1 d'une variable de type compteur sont aussi des affectations d'entier.

Exemple

```
horloge   H ;
compteur   C1, C2 ;
entier   A, B ;
tableau entier   X(1:3), Y(1:3, 1:2) ;
<H> plusun (C1),
    X $= sie A = 10 alors X + Y(, 1) sinon Y(, 2),
    Y $=  Y . Y ;
```

V.2 L'APPEL DE PROCEDURE

L'appel de procédure peut servir à charger des valeurs dans des variables de mémorisation. Il doit donc être effectué sous la portée d'une condition horloge, et à ce titre il est considéré comme une instruction de type affectation. Le mot clé appel, suivi du nom de la procédure, permet l'exécution d'une procédure écrite en FORTRAN ou en Assembleur.

Les paramètres effectifs, si la procédure a été déclarée avec un nombre de paramètres non nul, doivent tous apparaître. Les paramètres non scalaires doivent être transmis dans leur totalité. Le passage de paramètre se fait par nom, selon les conventions de FORTRAN.

Exemple

En reprenant les deux exemples du paragraphe III.4, les déclarations et les appels des procédures exposées s'écrivent :

```
horlogemère   H ;
entier   ADINS, CODEOP, OP1, OP2 ;
compteur   DIM ;
tableau entier   MESURE(0:255) ;
procédure   PROCHAIN(4) ;
fortran   HISTO(2) ;
   :
  <H>  appel PROCHAIN(ADINS, CODEOP, OP1, OP2), ... ;
   :
  <H>  appel HISTO(MESURE, DIM) ;
```

V.3 L'INSTRUCTION CONDITIONNELLE

Dans une description fonctionnelle, il est fondamental de pouvoir faire dépendre l'exécution de certaines instructions d'une condition portant sur des variables de types LASCAR. Nous avons donc défini une deuxième instruction conditionnelle, introduite par le mot clé sie. Par souci d'homogéneité, l'instruction conditionnelle sie est calquée sur l'instruction conditionnelle si de CASSANDRE. La condition qui est testée est donc une expression logique scalaire ou vectorielle. A chaque élément de cette expression est associée une liste d'instructions entre parenthèses dans la partie alors, et éventuellement une liste d'instructions dans la partie sinon, la partie sinon n'étant pas obligatoire. Les règles de compatibilité sont les mêmes que pour l'instruction conditionnelle si.

Exemple

L'exemple ci-dessous est à la fois une illustration de l'équivalence de deux unités, et de l'emploi de l'instruction conditionnelle sie dans un programme écrit en LASCAR.

Soit un comparateur 16 bits qui est construit à l'aide de 16 comparateurs élémentaires.

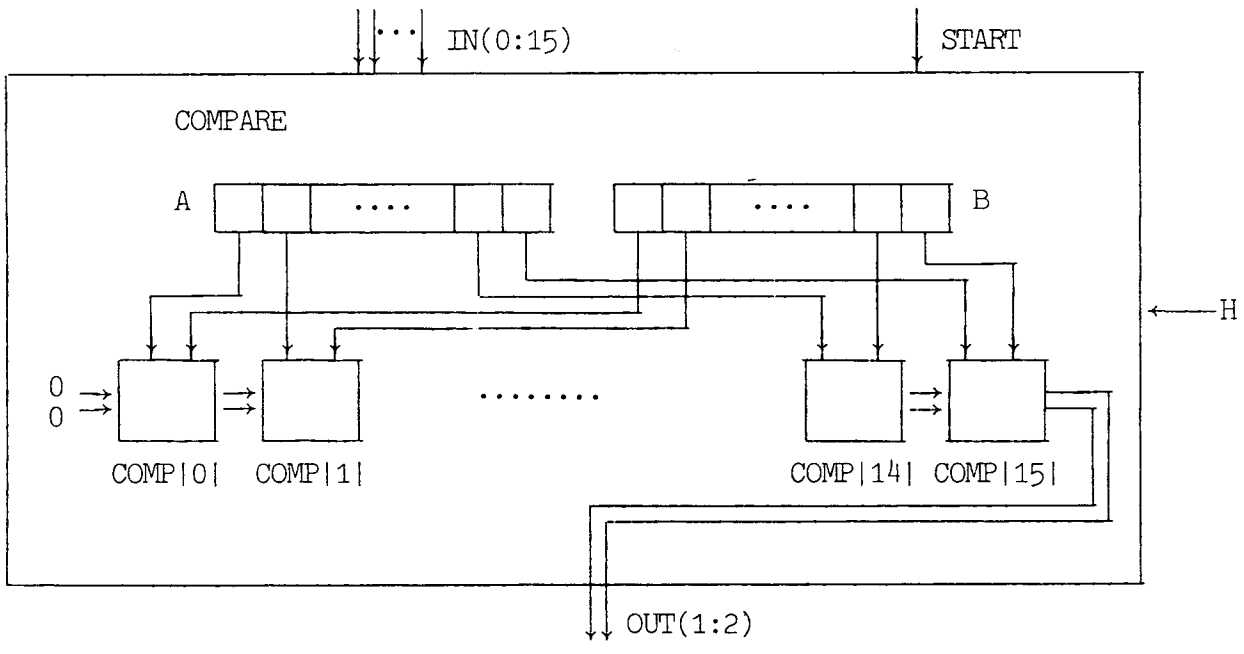
La comparaison est effectuée en 3 temps. Les deux premiers temps servent à charger dans les registres A et B les deux mots de 16 bits à comparer. Le troisième temps est utilisé pour effectuer l'opération. Les deux fils de sortie ont la signification suivante :

OUT(1) = 0 Les deux mots sont identiques
OUT(2) n'a alors pas de signification

OUT(1) = 1 Les deux mots sont différents.

OUT(2) = 0 Le deuxième élément de la comparaison est le plus grand

OUT(2) = 1 Le premier élément est le plus grand.



La description en CASSANDRE de ce comparateur est la suivante :

```
unité COMPARE(H, IN(0:15), START ; OUT(1:2)) ;
horlogemère  H ;
registre  A(0:15), B(0:15) ;
signal  U(1:2, 0:15) ;
externe  COMP (, , (1:2) ; (1:2)) ;
         "BOUCLE DE REPOS"
REPOS : <H> A <= IN,    "CHARGE LE 1ER ENTIER"
                si START alors allera INIT ;
INIT  : <H> B <= IN,    "CHARGE LE 2EME ENTIER"
                allera SUITE ;
                "CONNEXION DES COMPARATEURS ELEMENTAIRES"
SUITE : U(, 0) := 00 ;
        pourtout I de 0 a 14
        début  COMP|I|  (A(I), B(I), U(, I) ; U(, I + 1)) ; fin ;
               COMP|15| (A(15), B(15), U(, 15) ; OUT) ;
        <H> allera REPOS ;
unité  COMP  (E1, E2, X(1:2) ; Y(1:2)) ;
Y(1) := E1 ≠ E2 + X(1) ;
Y(2) := X(1).X(2) + -X(1).E1.-E2 ;
```

Le bon fonctionnement logique de l'unité COMPARE ayant été testé grâce à cette description, nous pouvons remplacer celle-ci par l'unité COMPAR, écrite en LASCAR, dont nous avons vérifié l'équivalence :

```
unité   COMPAR (H, IN(0:15), START ; OUT(1:2)) ;
horlogemère   H ;
entier   A, B ;
REPOS : <H> A $ = $(IN),
            si START alors allera INIT ;
INIT  : <H> B $ = $(IN), allera SUITE ;
SUITE :     sie A ≠ B alors OUT(1) := 1 sinon OUT(1) := 0 ;
            sie A > B alors OUT(2) := 1 sinon OUT(2) := 0 ;
        <H> allera REPOS ;
```

Sur les essais que nous avons effectués, le rapport des temps de simulation des unités COMPARE et COMPAR est de 50. L'emploi de l'unité COMPAR sera donc préféré lorsque l'on voudra simuler, sur un fonctionnement assez long, une machine qui contient ce comparateur 16 bits.

V.4 LES INSTRUCTIONS D'ITERATION

a) L'instruction "pourtout"

Nous avons vu que dans le langage CASSANDRE, l'instruction d'itération pourtout est utilisée pour décrire des structures répétitives. Cette instruction est considérée comme une macro, et l'ordre dans lequel l'index de boucle prend ses différentes valeurs est indifférent. Nous avons conservé la syntaxe et la sémantique de cette instruction inchangée dans LASCAR : tout se passe comme si les instructions à l'intérieur de la boucle étaient répétées autant de fois que l'index de boucle prend de valeurs différentes, l'ordre étant quelconque. Cette instruction est utilisée notamment pour des parcours de tables, ou pour effectuer des comptages et des prises de mesures, seul le nombre de fois où une condition est vérifiée étant dans ce cas intéressant.

Exemple

Supposons que nous simulons la machine IBM 360/370 qui contient 8 registres associatifs rapides pour accélérer la traduction adresse virtuelle-adresse réelle. A chacun de ces registres est joint un bit d'invalidité. Nous voulons par exemple déterminer l'histogramme du nombre de registres qui sont valides, au cours d'une simulation

```
horloge   H ;
registre  INVAL(1:8) ;
tableau entier  HISTO(0:8) ;
compteur  C ;
<H>  C $ = 0,
     pourtout I de 1 a 8
     début
     si INVAL(I) = 1 alors p1 (C)
     fin,
     HISTO(C) $ = HISTO(C) + 1 ;
```

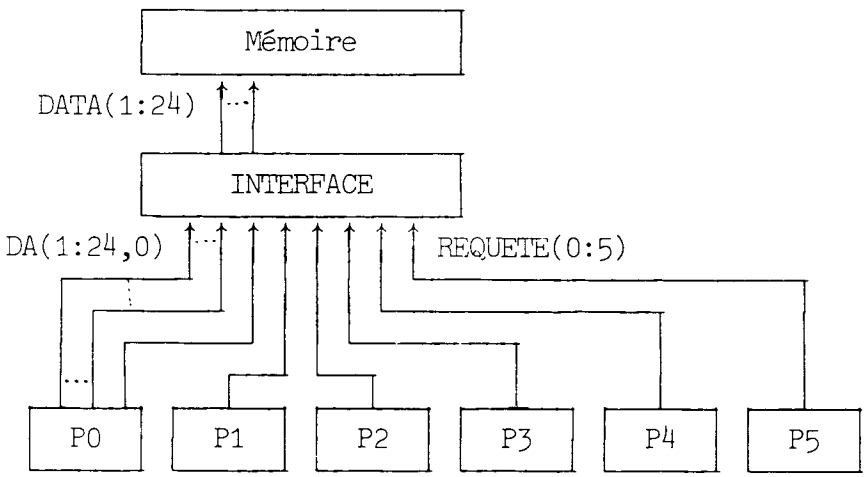
b) L'instruction "pour"

Nous avons eu cependant à régler le cas où, pour exprimer le déroulement d'un algorithme, l'ordre d'exécution des instructions qui sont à l'intérieur d'une boucle est important, et où notamment la suite des valeurs prises par l'index de boucle influe sur le résultat. Nous avons donc défini une deuxième instruction d'itération, introduite par le mot clé pour, dont la sémantique est identique à celle de la "boucle pour" d'Algol. L'index de boucle prend successivement toutes les valeurs comprises entre sa borne inférieure et sa borne supérieure, avec un incrément défini par le pas de la boucle (le pas pris par défaut est de 1).

Il est bien évident que cette nouvelle instruction, qui ne sert qu'à décrire des algorithmes, ne doit jamais être utilisée pour des descriptions de matériel. Aussi, dans l'instruction "pour", s'il est possible de tester des variables booléennes, il est interdit de modifier leur valeur : les seules affectations autorisées sont les affectations d'entiers, de tableaux entiers et de compteurs.

Exemple

Une mémoire est reliée à 6 processeurs identiques, qui ont une égale priorité. Si plusieurs processeurs veulent accéder simultanément à la mémoire, les conflits sont gérés par une interface qui effectue un balayage circulaire de gauche à droite des demandes, à partir de la dernière demande qui a été prise en compte.



Le signal REQUETE(I) est à 1 si le processeur numéro I veut accéder à la mémoire. L'entier PTR contient le numéro du dernier processeur qui a été servi. Le chemin de données, qui véhicule à la fois adresses et mots mémoire est de 24 bits.

Nous ne décrirons pas en détails toute l'unité d'interface, mais uniquement la séquence d'instructions qui décrit le mécanisme de sélection. TRAV est une variable de travail, TROUVE contient à la fin de la boucle le numéro de la prochaine demande à satisfaire.

```
unité   INTERFACE (H, REQUETE(0:5), DA(1:24, 0:5) ...
horlogemère  H ;
entier   PTR, TRAV, TROUVE ;
"on ne fait la sélection que s'il y a au moins une demande".
TEST : si /+ REQUETE alors faire SELECT ;
SELECT : <H> TROUVE $ = -1, "initialisation"
  pour  K = 1 a 6
  début      "on ne fait le calcul que si on n'a pas encore trouvé"
  sie TROUVE = -1 alors
      (TRAV $ = (PTR + K) rem 6,
      "TRAV est le numéro du prochain fil de REQUETE à tester"
      si  REQUETE(TRAV) alors TROUVE $ = TRAV)
  fin,
  PTR $ = TROUVE  "on ré-ajuste PTR"
  allera  CONNECT ;
CONNECT  :  "connexion entre la mémoire et le processeur"
   DATA  := DA(, TROUVE) ; ...
```

DEUXIEME PARTIE

UN EXEMPLE D'APPLICATION : MODELISATION D'UNE STRUCTURE MULTI-PROCESSEUR

I . PRESENTATION DU PROJET

La première application du langage LASCAR a été la modélisation, dans le cadre d'un contrat D.R.M.E., du projet multi-mini-processeur de la C.I.I., auquel nous collaborons. Cette étude, quoiqu'orientée vers une architecture très particulière, est basée sur des concepts extrêmement généraux, et sa présentation servira d'illustration des primitives du langage LASCAR.

La question à laquelle nous essayons de répondre est la suivante : est-il possible et rentable de construire une machine puissante à partir de mini-processeurs hautement intégrés, et donc de faible coût unitaire ?

Etant donné les performances des boitiers hautement intégrés actuellement disponibles, et la quantité de parallélisme qu'il est possible de mettre en évidence dans les programmes des utilisateurs [F14], [F15], [F17], le nombre de processeurs qu'il nous faut assembler pour construire un ordinateur très puissant se situe entre 20 et 100. Prenons cet ordre de grandeur comme hypothèse de travail. Il va sans dire que l'évaluation de l'organisation et de la performance d'une telle architecture est un des aspects fondamentaux du projet. L'absence de toute mesure sur un système en cours de spécification écarte la possibilité d'entreprendre un modèle mathématique du système. Une approche plus réaliste semble être un ensemble de simulations à différents niveaux.

Nous supposerons que le macro-code exécutable par le multi-processeur est une extension du code d'une machine existante. Ce choix est raisonnable, car il nous permet de récupérer des compilateurs, moyennant quelques modifications. En traçant ces compilateurs, nous pouvons donc obtenir des jeux d'essai pour le multi-processeur, qui nous permettront à la fois de vérifier son fonctionnement, et de comparer ses performances à celles de la machine existante, par simulation.

I.1 METHODOLOGIE D'EVALUATION

Une première étape consiste à décrire le processeur au niveau matériel, et à en simuler le fonctionnement sur chaque macro-instruction (le langage CASSANDRE a été utilisé à ce niveau).

L'intérêt de cette première simulation est double :

1/ vérifier la bonne conception de la logique et des micro-programmes,

2/ obtenir les intervalles de temps séparant les demandes de communication avec l'extérieur, pour les injecter dans les simulations successives.

La deuxième étape est une simulation, sur des traces d'instructions, d'un sous-ensemble du système. On peut envisager de considérer un faible nombre de processeurs, une mémoire locale et la mémoire principale. A ce niveau, les processeurs ne sont plus décrits en détail, mais apparaissent comme des "boites noires", dont seuls l'interface, et les actions engendrées par l'exécution des instructions sont connus. On fait des hypothèses simplificatrices sur l'influence des parties du système qui ne sont pas représentées (on peut choisir par exemple de les ignorer). Ce type de simulation permet de mesurer le ralentissement, sur le temps d'exécution des instructions, dû aux conflits d'accès à la mémoire. On peut alors tester différents algorithmes de gestion de la mémoire locale, l'influence de la taille des chemins de données, ainsi que des primitives système, sur différents jeux d'essai, et pour des nombres différents de processeurs.

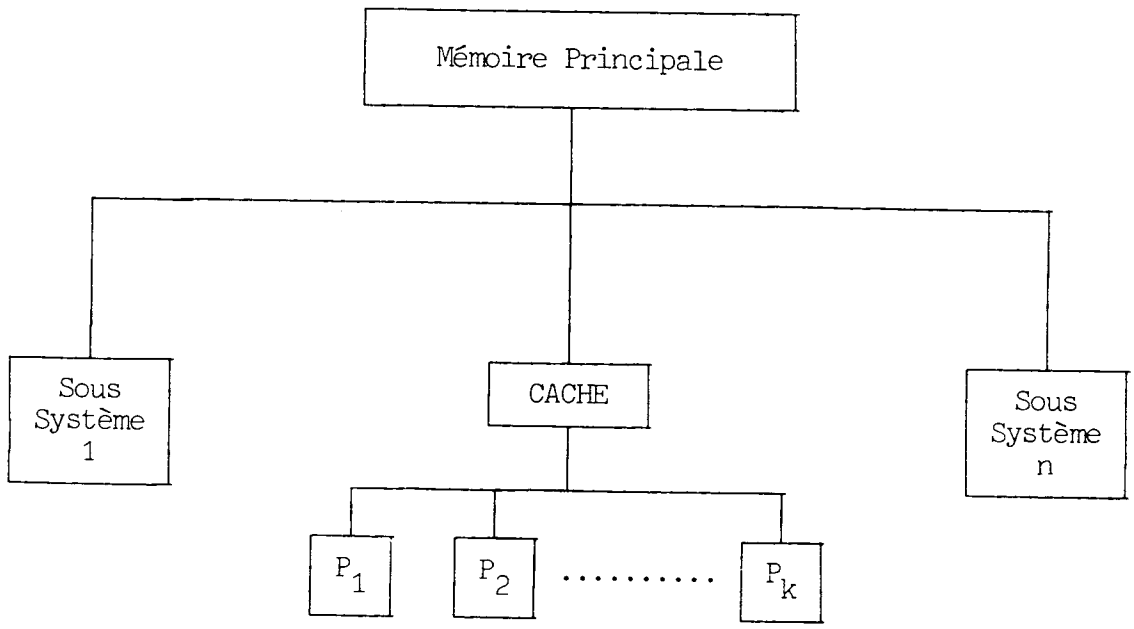
Un modèle plus abstrait, tenant compte de ces résultats, permettra d'évaluer le système tout entier, sur des traces d'instructions très longues. Une autre retombée du modèle global est de pouvoir corriger les hypothèses simplificatrices qui avaient été faites précédemment, de façon à simuler le modèle du deuxième niveau en ayant une meilleure connaissance de l'environnement du sous-système. Les résultats de cette nouvelle simulation serviront d'entrée pour une deuxième simulation du modèle le plus abstrait.

I.2 STRUCTURE DU SYSTEME

L'architecture du système que nous présentons ici a été définie par Madame RECOQUE et Messieurs CENSIER et MAZARE, notre contribution essentielle au projet concernant les aspects modélisation et évaluation.

Le multi-processeur est un ensemble de sous-systèmes (appelés "molécules" par l'équipe travaillant sur le projet, nous conserverons par la suite ce nom imagé) qui partagent une même mémoire principale. Chaque molécule est constituée de plusieurs processeurs identiques et d'une mémoire locale. A une molécule est attachée une fonction définissant le processeur logique réalisé par la molécule, c'est-à-dire le type de travail qui peut être traité par celle-ci. On définit ainsi les molécules d'entrées-sorties lentes (processeurs logiques lecteur de carte, imprimante, télétype), d'entrées-sorties rapides (processeurs logiques fichier sur disque, pagination), de calcul (processeurs logiques correspondant à divers macro-codes) etc...

Nous exposerons ici la deuxième étape de simulation, à l'aide du langage LASCAR, d'une molécule de calcul, composée de processeurs d'instruction émulant par micro-programmation le code IBM/360 [F11], reliés à une mémoire locale jouant le rôle d'un cache de la mémoire principale.



Sous-système i

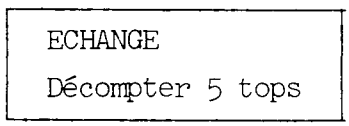
Le niveau de modélisation considéré est très fin, puisque nous prenons comme unité de temps la période de l'horloge du processeur. Cependant, les composants de la molécule ne sont décrits que du point de vue fonctionnel ; le lecteur trouvera dans [F7], rapport n° 2, Chapitre 3, les spécifications détaillées du matériel.

I.3 NOTATIONS

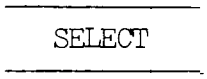
Nous adopterons, pour indiquer le temps de façon précise, les conventions d'écriture suivantes :

- Chaque boite rectangulaire représente un état. Sauf si on précise "décompter x tops", la boite représente une durée d'un cycle d'horloge.

- Un identificateur écrit en majuscule, en tête de la liste d'actions enfermées dans une boite, fait référence au nom de l'état correspondant dans le modèle.
  Exemple (tiré de PMEM) :

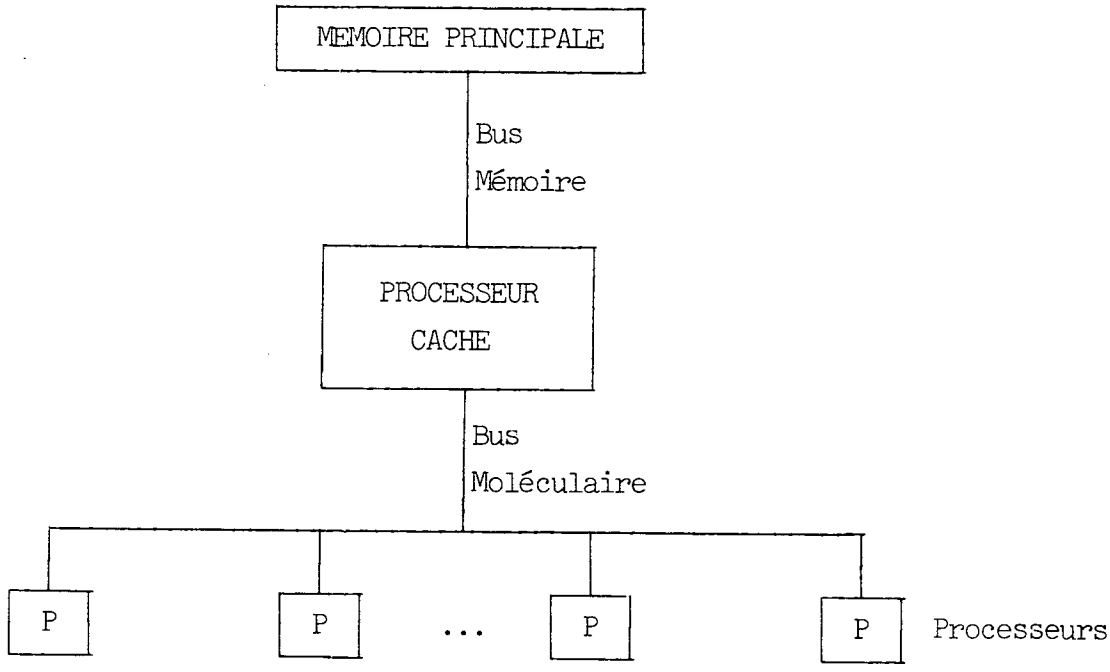


- Chaque boite aux bords arrondis représente un test câblé instantanné. Le passage dans une telle boite ne prend donc aucun temps.
  Exemple (tiré de PCACHE) :



II . ORGANISATION GÉNÉRALE DU MODÈLE

Un schéma très simplifié de la structure de la molécule est le suivant :



Cette structure est conservée dans la description en LASCAR que nous en avons faite. L'ensemble de la molécule est l'unité MODELE, elle-même constituée de trois sous-unités :

- PCACHE : modèle du processeur cache
- PMEM : modèle de la mémoire principale
- MICROP : modèle du processeur

PCACHE et PMEM apparaissent en un seul exemplaire. MICROP apparait en autant d'exemplaires que l'on veut simuler de processeurs dans la molécule : il est prévu de pouvoir connecter à PCACHE de une à douze duplications de MICROP.

Par souci d'optimisation, le bus moléculaire n'est pas représenté en tant que tel : chaque processeur est relié directement au cache, et les conflits d'accès se réglent à l'interface de celui-ci, ce qui ne change rien à la logique du programme, et évite de nombreuses variables intermédiaires.

De même, les registres, mémoires et chemins de données n'apparaissent pas dans la description. Les fils de connexion entre PMEM et PCACHE d'une part, PCACHE et MICROP d'autre part, ont seulement une signification logique, et sont uni-directionnels. Ils servent à demander une action (sorties de MICROP et PCACHE, entrées de PCACHE et PMEM), et à véhiculer les messages d'acquittement ou d'échec (entrées de MICROP et PCACHE, sorties de PMEM et PCACHE). Le modèle est synchronisé par une horlogemère de période 160 nano-secondes.

L'unité MODELE décrit essentiellement les connexions entre les sous-unités qui la composent :

. entre PMEM et PCACHE :

ECHMEM : 3 fils de demande d'échange
- ECHMEM(1) : lecture
- ECHMEM(2) : écriture
- ECHMEM(3) : test and set (lecture et ré-écriture).

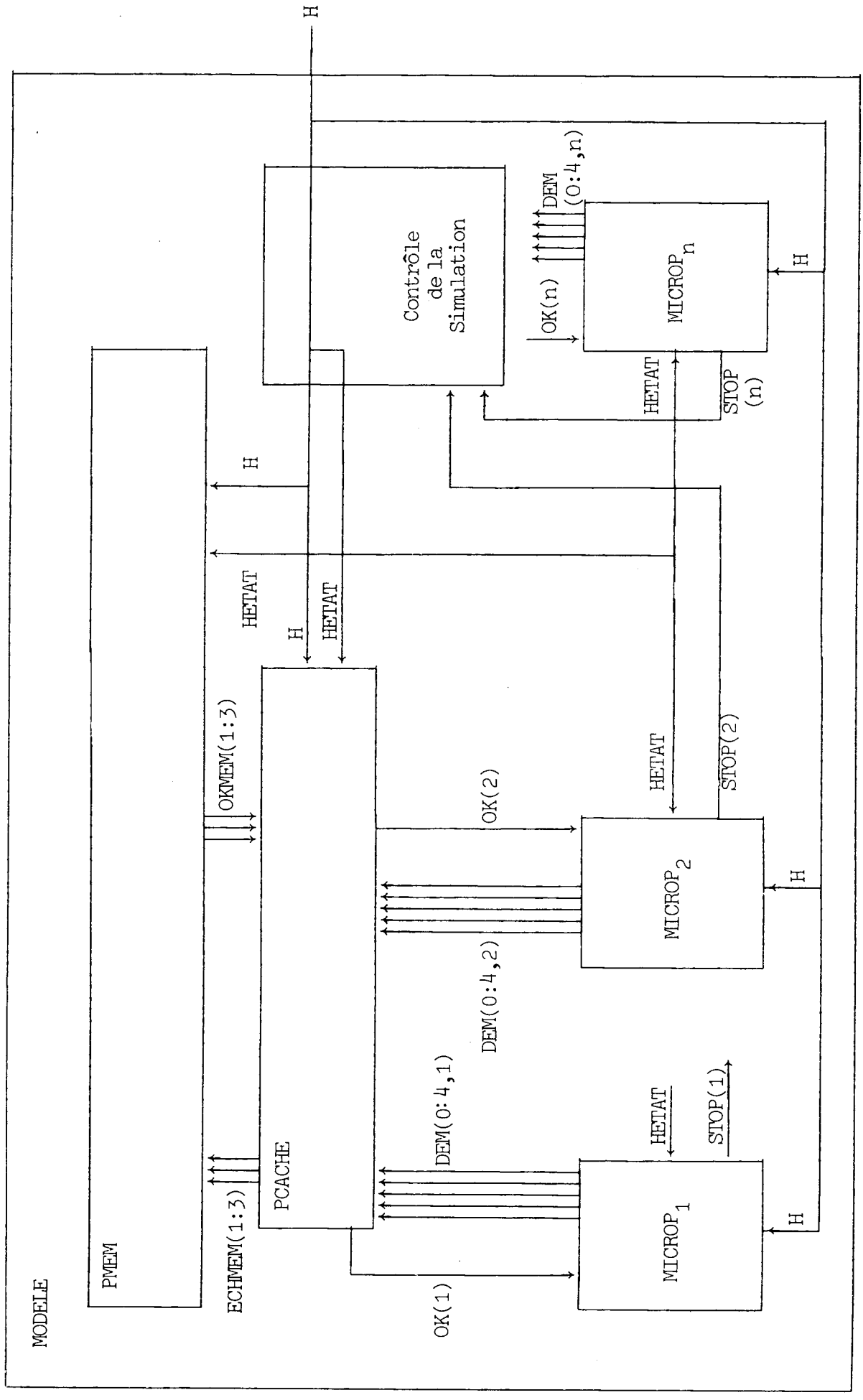
OKMEM : 3 fils d'acquittement
- OKMEM(1) : donnée disponible
- OKMEM(2) : donnée écrite
- OKMEM(3) : donnée disponible et mémoire en attente de ré-écriture.

. entre PCACHE et chaque processeur MICROP :

DEM : 5 fils de demande au cache
- DEM(0) : lecture
- DEM(1) : écriture
- DEM(2) : test and set
- DEM(3) : invalidation partielle
- DEM(4) : invalidation totale

OK : 1 fil d'acquittement de la demande.

L'unité MODELE comporte en plus quelques instructions de contrôle, qui servent essentiellement à commander au cache et aux processeurs l'impression de statistiques en leur envoyant une impulsion HETAT lorsque un processeur arrive en fin d'exécution de trace (signal STOP). Lorsque tous les processeurs se sont arrêtés, la procédure FINSIM commande l'arrêt de la simulation.


MODELE
PMEM
PCACHE
MICROP 1
MICROP 2
MICROP n
Contrôle de la Simulation
H
HETAT
STOP(1)
STOP(2)
STOP (n)
OK(1)
OK(2)
OK(n)
OKMEM(1:3)
ECHMEM(1:3)
DEM(0:4,1)
DEM(0:4,2)
DEM (0:4,n)

Description en LASCAR

```
U. S.   TEXTE SOURCE

00001   'UNITE' MODELE (H;);
00013   'HORLOGEMERE' H;
00016   'HORLOGE' METAT;  "POUR SORTIR ETAT DES PROCESSEURS"
00023   'HORLOGE' FIN;   "COMMANDE FIN DE SIMULATION"
00028   'SIGNAL' STOP (1:4);          "SIGNAL ARRET PROCESSEUR (I)"
00039   'COMPTEUR' C1,C2,ARRET;
00052   'SIGNAL' DEM(0:4,1:12),OK(1:12),ECHMEM(1:3),OKMEM(1:3);
00099   'EXTERNE' PCACHE('HORLOGE','HORLOGE',(0:4,1:12),(1:3);(1:3),(1:12));
00142   'EXTERNE' PMEM('HORLOGE','HORLOGE',(1:3);(1:3));
00165   'EXTERNE' MICROP('HORLOGE','HORLOGE',;,,,,,);
00185   'PROCEDURE' FINSIM;
00193   PCACHE(d,METAT,DEM,OKMEM;ECHMEM,OK);
00229   PMEM(d,METAT,ECHMEM;OKMEM);
00256   'POUR' I=1 'A' 4
00262   'DEBUT'
00263   MICROP(I)  (d,METAT,OK(I);STOP(I),DEM(0,I),
00304              DEM(1,I),DEM(2,I),DEM(3,I),DEM(4,I));
00341   'FIN';
00343   <FIN>  'H1'(ARRET), 'SIE' ARRET=0 'ALORS'
00366                              ('CALL' FINSIM);
00376   <H>  C1$=C2, C2$=$(STOP(1:2)),
00401   'SIE' C2 > C1 'ALORS' METAT:=1 'SINON' METAT:=0,
00424   'SI' (/.STOP) 'ALORS' (FIN:=1);
00442
```

III . LE PROCESSEUR : MICROP

III.1 MACRO-CODE

Le processeur interprète du code IBM/360. Nous extrayons de [F7, rapport n° 2, Chapitre 3] la présentation du processeur envisagé par Monsieur CENSIER :

> "La complexité du code 360 et le nombre important des registres nécessaires ne permettent pas d'envisager que le processeur d'exécution de ce code puisse être intégré sur un seul morceau de silicium. (...) Aussi le processeur d'instruction est-il envisagé sous la forme d'un nombre limité de boitiers pour lesquels le niveau d'intégration est, au moment considéré, le plus élevé qu'il soit possible d'obtenir sans dégradation de performance. Le niveau de performance de référence est celui qu'il est possible d'cbtenir en faisant appel à la technologie TTL disponible actuellement".

Nous simulons son fonctionnement sur une trace d'exécution d'un programme de mise à jour d'une base de données, obtenue par interprétation des instructions générées par le compilateur SOCRATE, à l'aide de l'outil PILOTE. Cette trace, dont le parallèlisme a été exhibé au niveau système, est une portion de la trace sur laquelle Monsieur MAZARE a effectué des simulations moins détaillées [F7, rapport n° 3, Chapitre 2]. Nous estimons que c'est un jeu d'essai réaliste pour étudier le comportement de la molécule.

Le processeur est micro-programmé.
Nous avons écrit avec Madame RECOQUE une première version des micro-programmes qui interprètent les instructions de registre à registre, et de registre à mémoire du code 360. Les instructions longues de mémoire à mémoire et les instructions en virgule flottante n'ont pas été traitées en raison de leur complexité. Nous ne prétendons donc pas avoir ici une description exacte du fonctionnement du processeur, mais une estimation raisonnable de son comportement sur les instructions les plus fréquemment utilisées. Nous appelons signature d'une instruction la suite : nombre de micro-instructions, lecture ou écriture, nombre de micro-instructions, lecture ou écriture, etc..., qui compose son micro-programme d'interprétation. Les signatures des instructions que nous avons traitées sont données en annexe II.

La simulation de l'exécution d'une instruction est donc, pour notre modèle, une suite de lectures et/ou écritures, et de cycles de décomptage, une micro-instruction prenant deux cycles d'horloge. Les instructions ont été regroupées en 10 types, en fonction de la structure de leur signature. La lecture de la prochaine instruction à traiter se fait par l'appel de la procédure NEXTI, qui effectue, à partir de la trace enregistrée sur fichier, un premier décodage et fournit au modèle les seules caractéristiques significatives :

- adresse instruction
- longueur (en nombre de demi-mots)
- type
- temps d'exécution (en nombre de cycles d'horloge) : trois paramètres.

III.2 ALGORITHME D'APPEL DES INSTRUCTIONS

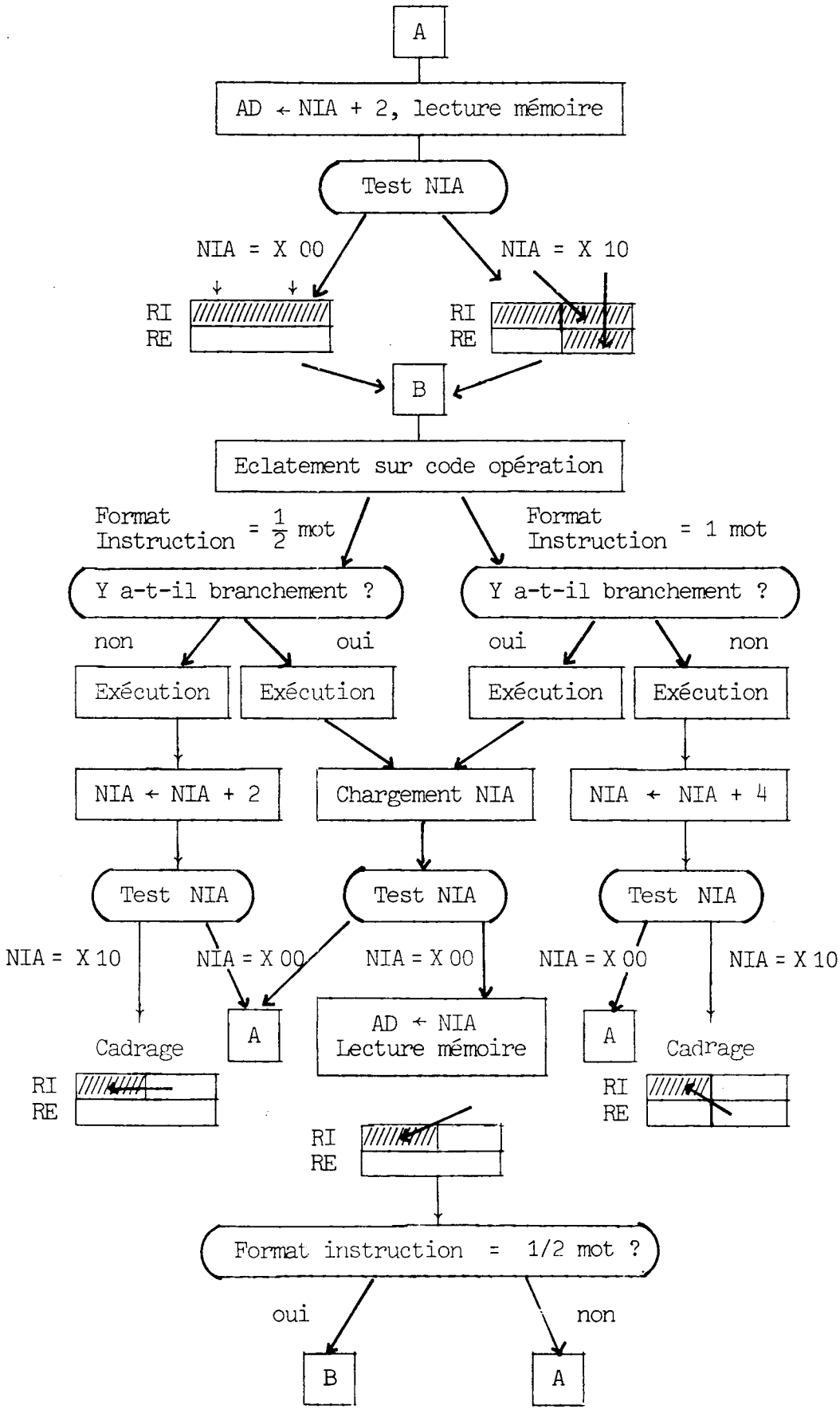
Les instructions considérées occupent en mémoire une longueur variable : 1/2 mot pour les instructions dont les deux opérandes sont des registres, 1 mot pour les instructions dont un opérande est un élément mémoire ou est immédiat (le mot est de 4 octets).

Le processeur dispose de plusieurs registres de longueur 1 mot :

- RI : registre contenant l'instruction
- RE : extension de RI
- NIA : adresse de la prochaine instruction à exécuter (compteur ordinal)
- AD : adresse du mot à lire ou à écrire, lors d'un échange avec la mémoire.

Les communications avec le cache se font par un chemin de données de 4 octets. Les deux bits de poids faible de AD sont ignorés en lecture : on lit toujours 4 octets, cadrés sur une frontière de mot.

Suivant que l'instruction précédente avait une longueur de 2 ou 4 octets, tout ou partie de l'instruction courante peut être déjà disponible dans le processeur, ou au contraire il faut la lire entièrement. Monsieur CENSIER a élaboré l'algorithme d'appel des instructions, dont nous donnons ci-après le sous-ensemble concernant les seules instructions de longueur inférieure ou égale à 1 mot.

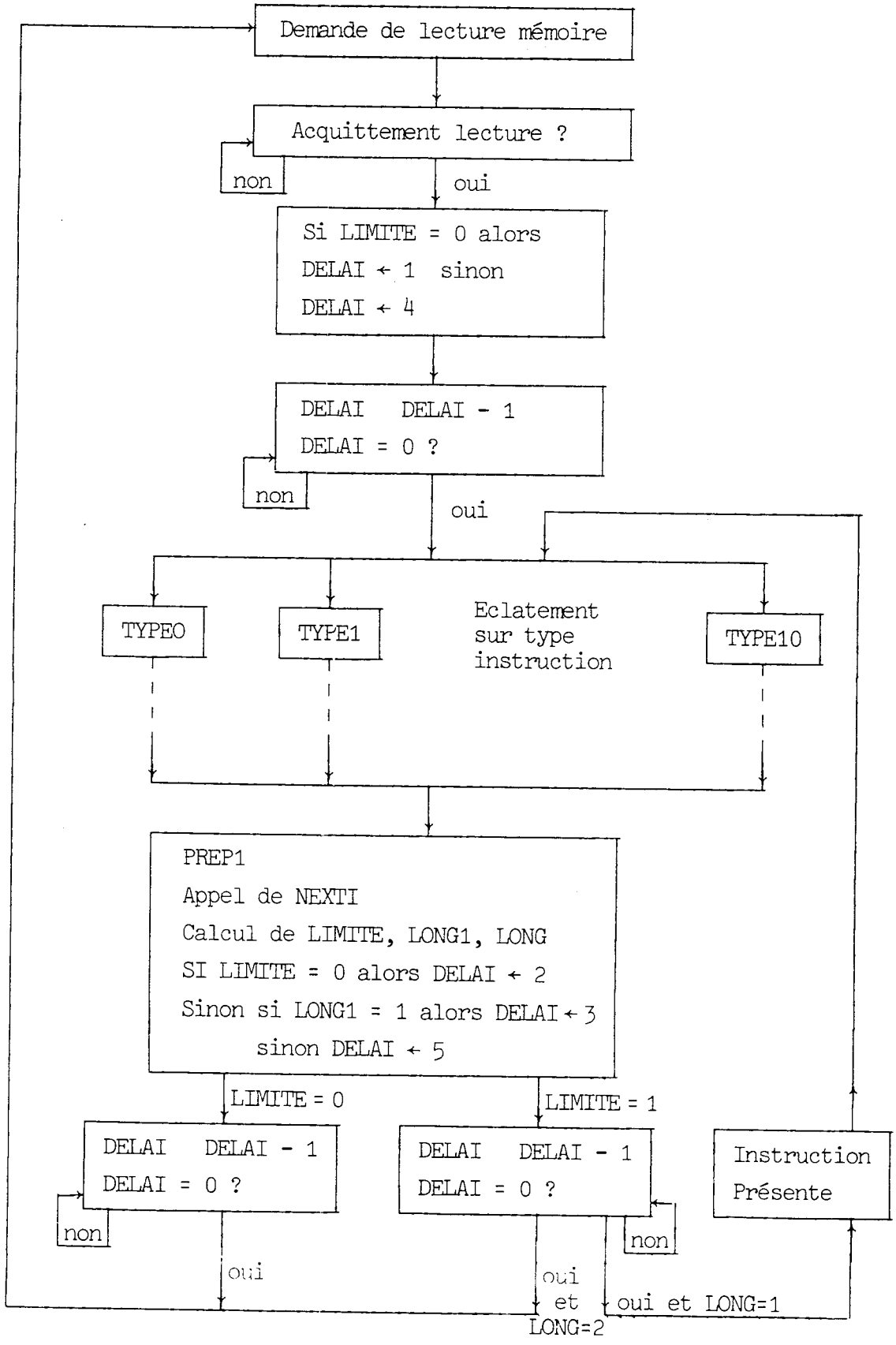

A
AD ← NIA + 2, lecture mémoire
Test NIA
NIA = X 00
NIA = X 10
RI
RE
RI
RE
B
Eclatement sur code opération
Format Instruction = 1/2 mot
Format Instruction = 1 mot
Y a-t-il branchement ?
Y a-t-il branchement ?
non
oui
oui
non
Exécution
Exécution
Exécution
Exécution
NIA ← NIA + 2
Chargement NIA
NIA ← NIA + 4
Test NIA
Test NIA
Test NIA
NIA = X 10
NIA = X 00
NIA = X 00
NIA = X 00
NIA = X 10
Cadrage
A
AD ← NIA
Lecture mémoire
A
Cadrage
RI
RE
RI
RE
RI
RE
Format instruction = 1/2 mot ?
oui
non
B
A

Cet algorithme est cependant inutilement précis, pour une simulation du processeur au cours de laquelle les seuls évènements intéressants sont les échanges avec l'extérieur. Nous en avons donc extrait un algorithme simplifié. L'éclatement sur code opération se fait en fonction du type de l'instruction, tel qu'il est renvoyé par la routine NEXTI.

D'autre part :

- DELAI est un compteur qui sert à introduire des retards.
- LIMITE = 0 si l'instruction en cours de traitement est cadrée sur une frontière de mot, 1 sinon.
- LONG1 est la longueur de l'instruction qui vient d'être traitée, LONG la longueur de la nouvelle instruction, en nombre de demi-mots.

Nous avons donc simulé l'algorithme suivant :


Demande de lecture mémoire
Acquittement lecture ?
non
oui
Si LIMITE = 0 alors
DELAI ← 1 sinon
DELAI ← 4
DELAI DELAI - 1
DELAI = 0 ?
non
oui
TYPE0
TYPE1
Eclatement
sur type
instruction
TYPE10
PREP1
Appel de NEXTI
Calcul de LIMITE, LONG1, LONG
SI LIMITE = 0 alors DELAI ← 2
Sinon si LONG1 = 1 alors DELAI ← 3
sinon DELAI ← 5
LIMITE = 0
LIMITE = 1
DELAI DELAI - 1
DELAI = 0 ?
non
oui
DELAI DELAI - 1
DELAI = 0 ?
non
oui
et
LONG=2
oui et LONG=1
Instruction
Présente

Les instructions suivantes, extraites de l'unite MICROP, sont la description en LASCAR de l'algorithme d'appel des instructions.

Description en LASCAR:

```
U. S.   TEXTE SOURCE

00001   'UNITE' MICROP (H,HETAT,OK;STOP,LEC,ECR,TS,INV,RAZ);
00045   'HORLOGEMERE' H,HETAT;
00054   'COMPTEUR' NUMPROC, "NUMERO DU PROCESSEUR"
00063              SING,SINGL,SINGE,   "POINTS SINGULIERS"
00080           K,      "COMPTEUR PARTIEL D ATTENTE"
00082                DELAI,  "COMPTEUR DE CYCLES"
00088            INTPS,      "TEMPS SEPARANT 2 APPELS MEMOIRE CONSECUTIFS"
00094                 CICLES,        "COMPTEUR DE CYCLES"
00101                NBINS,      "COMPTEUR NOMBRE D'INSTRUCTIONS"
00107                KREPOS,      "COMPTEUR CYCLES INACTIVITE"
00114              KLECI,   "COMPTEUR LECTURES INSTRUCTIONS"
00120            ALECI,  "ATTENTE LECTURE INSTRUCTION"
00174   'ETAT' ETX(0:12);   "ADRESSES BRANCHEMENT EN FONCTION DES CODE OP."
00185   'ENTIER' ADINS, "ADRESSE INSTRUCTION"
00192            TYPE,  "TYPE INSTRUCTION"
00197            EXEC1,EXEC2,EXEC3 ,  "TEMPS EXEC DE L'INSTRUCTION"
00215            LIMITE, "CADRAGE DE L'INSTRUCTION"
00222            LONG1, LONG;   "LONGUEUR DE L'INSTRUCTION"
00233   'TABLEAU' 'ENTIER' HISTOGR(1:64),     "HISTOGRAMME DES TEMPS
00249                                        SEPARANT 2 APPELS MEMOIRE CONSECUTIFS"
00249                  RALEC(0:63),     "HISTOGRAMME DES ATTENTES EN LECTURE"
00261                  RAECR(0:63);     "HISTOGRAMME DES ATTENTES EN ECRITURE
00273   'TABLEAU' 'ENTIER' INSTYP(0:10);      "NOMBRE D'APPELS DES DIFFERENTS
00288                                       TYPES D'INSTRUCTIONS"
00288   'PROCEDURE' NEXTI(7);      "VA CHERCHER LA PROCHAINE INSTRUCTION"
00298   'FORTRAN' PINSTY(1);      "IMPRIME INSTYP"
00309   'FORTRAN' STAT(14),ETATP(13),SIMPC(2),HISTO(2),HISTOL(2),HISTOE(2);
00367      "SOUS-PROGRAMMES D'IMPRESSION DE STATISTIQUES"

00485   LECTI1: <H>    'P1'(KLECI),'P1' (INTPS),  'ALLERA' LECTI11;
00522   LECTI11: LEC:=1; 'FAIRE' HISTOIRE; <H> 'ALLERA' LECTI12;
00558   LECTI12: 'SI' OK 'ALORS' (LEC:=0; 'FAIRE' HISLEC);
00586            <H> 'SI' OK 'ALORS' ('ALLERA' TESTNIA1, 'P1'(INTPS))
00613         'SINON' ('P1'(ALECI), 'P1'(K));
00630   TESTNIA1: <H> 'SIE' LIMITE = 0  'ALORS' DELAI $=1
00659                          'SINON' DELAI $=4,
00668                                  'P1' (INTPS),
00677                   'ALLERA' ATTENTE;
00686   ATTENTE: <H> 'M1'(DELAI),
00706                                  'P1' (INTPS),
00715                 'SIE' DELAI = 0 'ALORS' 'ALLERA' CODEOP;
00732   CODEOP: <H>    'ALLERA' ETX(TYPE), 'P1' (INTPS),
00762               INSTYP(TYPE) $=INSTYP(TYPE)+1;
```

```
01149   PREP1: <H> LONG1 $= LONG, 'P1'(NBINS),
01178                'CALL' SIMPC(CICLES,NUMPROC),    "IMPRESSION D'UNE TRACE
01201        'CALL' NEXTI(NUMPROC,TYPE,ADINS,LONG,EXEC1,EXEC2,EXEC3),
01251                        "LECTURE PROCHAINE INSTRUCTION SUR UN FICHIER"
01251             LIMITE $= ADINS 'REM' 4,      "CADRAGE INSTRUCTION"
01266                    'P1' (INTPS),
01275          'SIE' LIMITE=0 'ALORS' (DELAI $= 2, 'ALLERA' VERSA)
01301          'SINON' ('SIE' LONG=1 'ALORS' (DELAI $= 3, 'ALLERA' VERSD)
01328                    'SINON' (DELAI $=5, 'ALLERA' VERSD));
01347   VERSA: <H> 'M1'(DELAI),'P1'(INTPS), 'SIE' DELAI=0 'ALORS' 'ALLERA'
01384              LECTI1;
01391   VERSD: <H> 'M1'(DELAI),'P1'(INTPS), 'SIE' DELAI=0 'ALORS'
01427                    ('SIE' LONG=1 'ALORS' 'ALLERA' VERSB
01442                                  'SINON' 'ALLERA' LECTI1);
01452   VERSB: <H> 'P1'(INTPS), 'ALLERA' CODEOP;


03554   HISTOIRE:<H> 'SIE' INTPS > 64 'ALORS'
03576                'P1'(SING)
03683                                 'SINON'
03684                HISTOGR(INTPS) $=HISTOGR(INTPS)+1, INTPS$=0;
03724   HISLEC: <H>   'SIE' K > 63 'ALORS'
03740                'P1'(SINGL)  'SINON'
03749           RALEC(K) $=RALEC(K)+1, K$=0;
```

III.3 EXECUTION DE L'INSTRUCTION

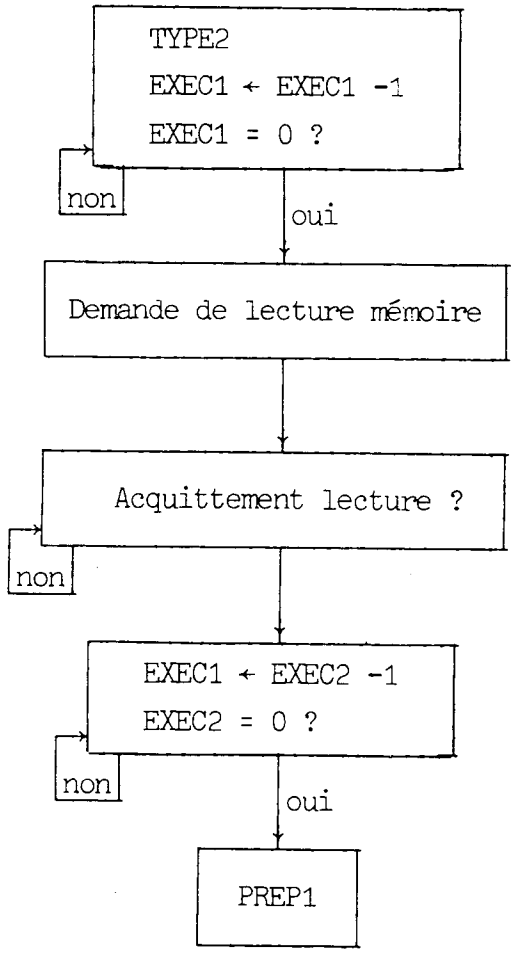
Nous ne détaillerons pas ici tous les types mais nous nous contenterons d'illustrer le fonctionnement du modèle sur un exemple.

Considérons le type n° 2.

Ce sont les instructions de signature :

EXEC 1 * EXEC 2

c'est-à-dire : calcul adresse opérande
lecture opérande
exécution



L'exécution d'une instruction du type n°2 est decrite en LASCAR par le morceau de programme suivant, extrait de MICROP:

```
01478   TYPE2: <H> 'P1'(INTPS), 'M1'(EXEC1),
01505                'SIE' EXEC1=0 'ALORS'
01514                        ('P1'(KLECOP),  'ALLERA' TYPE2L);
01534   TYPE2L: LEC:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST2L;
01568   TEST2L: 'SI' OK 'ALORS'
01579                     (LEC:=0; 'FAIRE' HISLEC;
01594                     <H> 'P1'(INTPS),  'SIE' EXEC2=0 'ALORS' 'ALLERA'
01621                        'SINON' 'ALLERA' SUITEX2)
01631              'SINON' (<H>  'P1'(ALECOP), 'P1'(K));
01652   SUITEX2: <H>  'M1'(EXEC2),
01672                              'P1' (INTPS),
01581                 'SIE' EXEC2=0 'ALORS' 'ALLERA' PREP1;
```

III.4 MESURES RECUEILLIES DANS LE PROCESSEUR

Les mesures que l'on prélève lors de la simulation du processeur sont de deux catégories :

- des mesures qui caractérisent la trace :
  - nombre de lectures instructions (KLECI)
  - nombre de lectures opérandes (KLECOP)
  - nombre d'écritures (KECROP)
  - nombre de test and set (KTES)
  - histogramme des temps séparant deux échanges avec le cache, temps d'attente non compris (HISTOGR et SING)

- des mesures qui sont des résultats de la simulation :
  - histogramme des temps d'attente sur lecture (RALEC et SINGL)
  - histogramme des temps d'attente sur écriture (RAECR et SINGE)
  - temps moyen d'attente sur test and set (ATES)

Lorsque le processeur arrive en fin de trace, il envoie au cache un ordre d'invalidation partielle (NEXTI, rencontrant une fin de fichier, renvoie le type 10).
Après réception du message d'acquittement de l'invalidation, le processeur imprime toutes les mesures (routines Fortran STAT, HISTO, HISTOL, HISTOE, SIMPC), et positionne le signal STOP. C'est ce signal qui, recueilli par l'organe de contrôle de l'unité MODELE, génère l'impulsion HETAT, laquelle commande au cache et à tous les autres processeurs encore actifs l'impression des statistiques qui les concernent. Puis le processeur se met en état d'inactivité (état HALT), et n'influe plus sur les autres unités du modèle.
L'arrêt du processeur se traduit par les instructions suivantes :

```
03822   REPOS: LEC:=0;
03834          ECR:=0;
03840          TS:=0;
03845          INV:=0;
03851          RAZ:=0;
03857     <H> 'CALL' STAT(NUMPROC,NBINS,KREPOS,KLECI,ALECI,KLECOP,
03906          ALECOP,KECROP,AECROP,KTES,ATES,KINV,AINV,EXEC1),
03954   'CALL' SIMPC(CICLES,NUMPROC),
03977   'CALL' HISTO(HISTOGR,SING),
03998   'CALL' HISTOL(RALEC,SINGL),
04019   'CALL' HISTOE(RAECR,SINGE),
04040   'CALL' PINSTY(INSTYP),
04056   'ALLERA' STOPM;
04063   STOPM: STOP:=1;
04076   <H>  'ALLERA' HALT;
04085   HALT: ;
```

Le programme complet correspondant à l'unité MICROP est donné en annexe III.

IV . LE PROCESSEUR CACHE : PCACHE

IV.I PRINCIPES DE FONCTIONNEMENT

Soit 2^M la taille, en octets, de la mémoire principale, et 2^m celle de la mémoire cache. Ces deux mémoires sont divisées en blocs de 2^b octets. Lorsqu'un processeur veut lire un mot qui est absent du cache, on lit en mémoire centrale, et on met dans la mémoire cache, le bloc qui contient ce mot.

La mémoire principale est (logiquement) découpée en 2^L lignes et 2^C colonnes de blocs, de telle sorte que :

$L + C + b = M$

La mémoire cache est découpée en 2^L lignes et 2^x colonnes de blocs. On a donc :

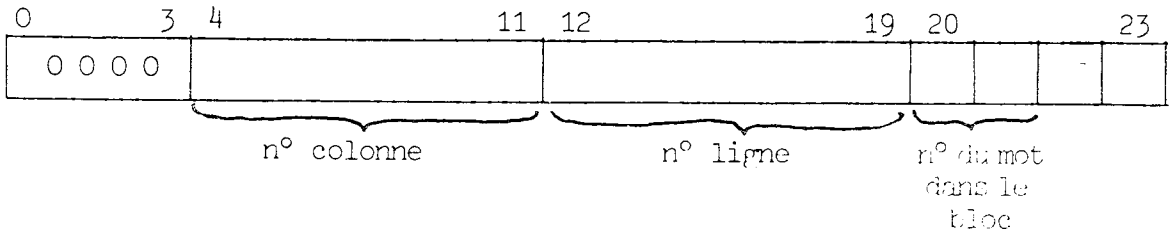
$L + x + b = m$

Le rapport entre les tailles des deux mémoires est donc aussi le rapport entre leurs nombres de colonnes.

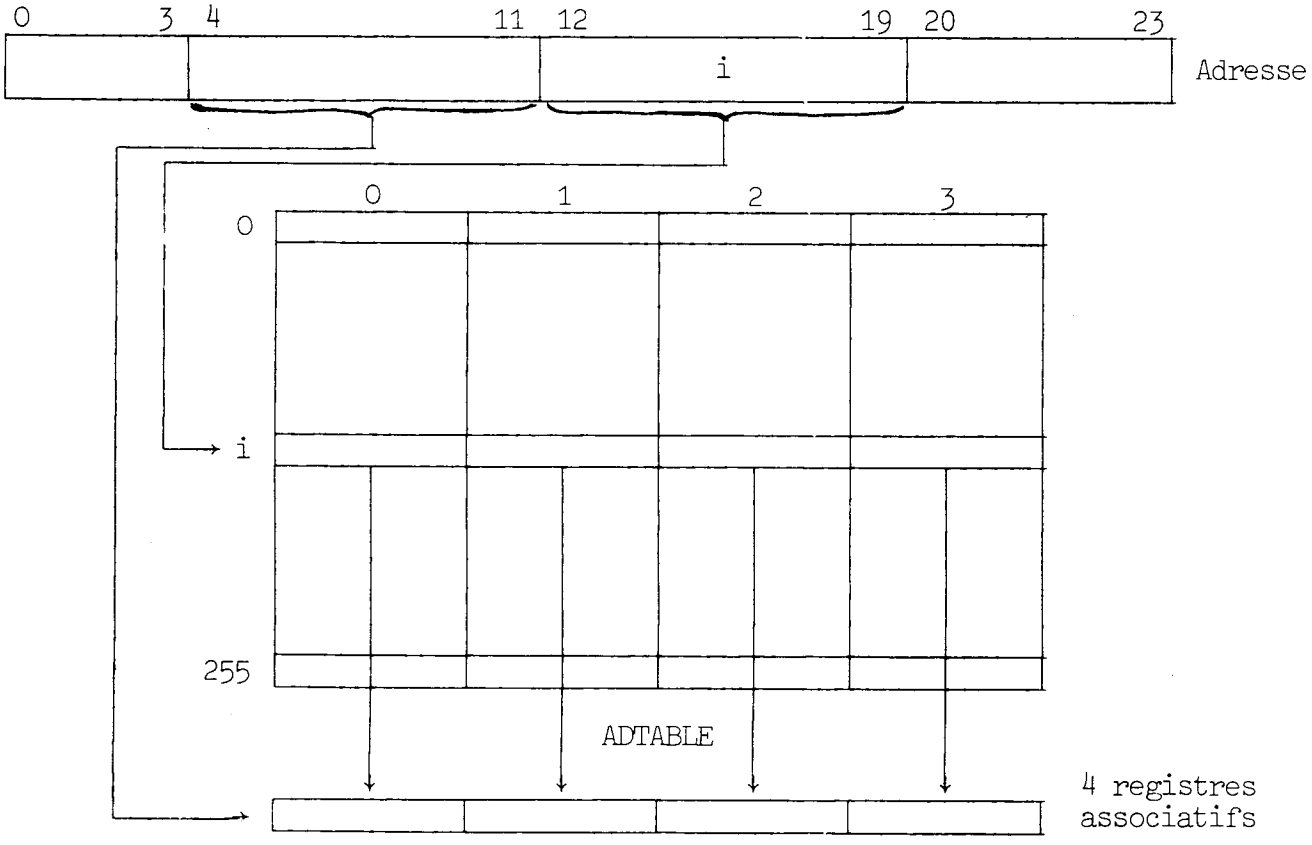
Pour fixer les idées, considérons une mémoire principale de 1024 K octets, et un cache de 16 K octets, découpés en blocs de 16 octets répartis sur 256 lignes. Nous avons donc :

$L = 8$ $\quad M = 20$
$C = 8$ $\quad m = 14$
$x = 2$
$b = 4$

Une adresse, dans le macro-code 360, tient sur 24 bits. Cette adresse est découpée en champs. Les bits de poids forts indiquant le numéro de la colonne, et les bits centraux le numéro de ligne du bloc qui contient le mot référencé. Pour l'exemple ci-dessus, la taille de la mémoire principale étant un méga octets, les quatre premiers bits de l'adresse sont toujours nuls.



L'organisation du processeur estinspirée de celle de caches existants [F10], [F18]. Le processeur cache contient une table d'adresses ADTABLE de 256 lignes et 4 colonnes. Lorsqu'un bloc est référencé, les bits 12 à 19 de l'adresse servent à indexer ADTABLE. Le bloc est dans le cache si l'une des 4 cellules ainsi indéxées contient son numéro de colonne (bits 4 à 11).



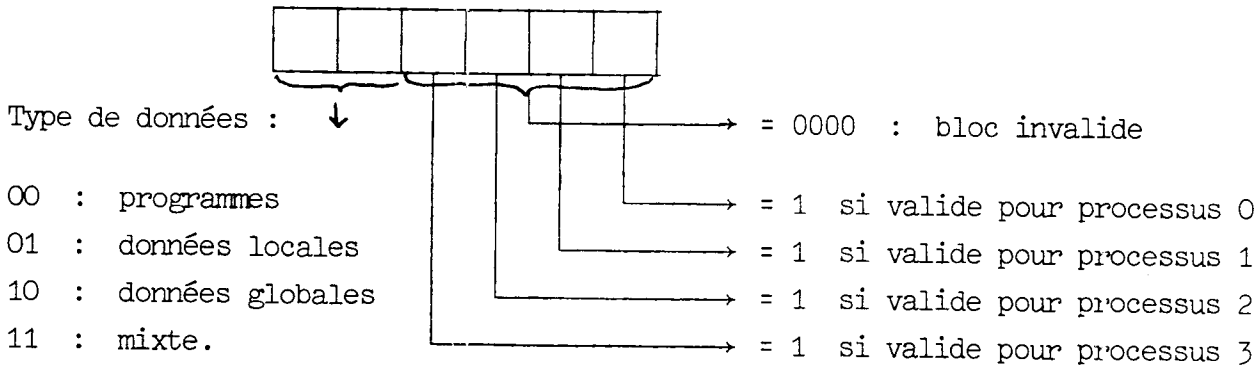
Pour accélérer la recherche d'un bloc, on adjoint à ADTABLE 4 registres associatifs. Ces registres sont chargés en parallèle par une ligne de ADTABLE. La comparaison se fait associativement entre le numéro de colonne de l'adresse envoyée par le processeur, et les quatre registres.

Nous avons participé à la définition fonctionnelle du cache. L'étude de sa structure matérielle et du séquencement des algorithmes a été réalisée par Monsieur CENSIER, et le lecteur en trouvera tous les détails dans [F7, rapport N° 2]. Il nous parait cependant utile, pour la compréhension du modèle, d'en rappeler ici les caractéristiques principales.

Pour des raisons de cohérence des informations entre les caches de différentes molécules et la mémoire principale, dans le cadre du système global [F7 rapport n° 2, Chapitre 3], tout processeur qui demande un échange mémoire envoie, en même temps que l'adresse, un préfixe Z identifiant le type de donnée référencée (donnée globale au système, donnée locale au processus, programme) et le numéro du processus (numéro local à la molécule).

Le cache contient une table NUMZ, de même nombre de lignes et de colonnes que ADTABLE, qui indique, pour chaque bloc, le type de données qu'il contient et les numéros de processus pour lesquels il est valide. Un bloc peut donc être partagé entre plusieurs processus.

Pour un cache relié à quatre processeurs, les conventions d'écriture d'un élément de NUMZ sont les suivantes :



La comparaison entre le préfixe Z du processeur et l'élément correspondant de NUMZ doit être positive pour qu'un bloc, identifié comme présent dans ADTABLE, soit accessible au processeur. Dans le cas contraire, un échange préalable avec la mémoire principale est nécessaire.

En nous basant sur les résultats d'autres études [F18], il a été choisi d'appliquer, pour le remplacement des blocs l'algorithme LRU, limité aux 4 blocs d'une même ligne de la table d'adresses.

IV.2 ACTIVATION DU PROCESSEUR CACHE

Le cache est sollicité de la part des processeurs, pour l'exécution d'une parmi cinq actions possibles :

a) Demande de lecture

Une demande de lecture est satisfaite immédiatement si la donnée est présente dans le cache. Dans ce cas, elle prend deux cycles d'horloge : un cycle de test de présence, et un cycle de lecture. Cependant, les circuits effectuant ces deux opérations travaillant en parallèle, il peut y avoir recouvrement pour deux demandes venant de deux processeurs différents (structure en "pipe-line"). Le cache est donc capable de renvoyer une donnée à chaque cycle d'horloge, dans le cas idéal où tous les processeurs ne font que des demandes de lectures de données présentes dans le cache. Par contre, si la donnée est absente, un échange préalable avec la mémoire principale, pour amener en mémoire cache le bloc qui la contient, est nécessaire.

b) Demande d'écriture

Toute écriture est exécutée immédiatement en mémoire principale. Si le bloc contenant la donnée à écrire est présent dans le cache, l'écriture est effectuée à la fois dans le cache et en mémoire principale ; dans le cas contraire, seule l'écriture en mémoire principale est effectuée, de façon à éviter une lecture préalable du bloc. Une écriture demande donc toujours un échange avec la mémoire principale.

c) Demande de test and set

Dans le système considéré, tout processus voulant accéder à une ressource ou à une donnée globale partagée doit préalablement effectuer un "test and set" sur la variable globale servant à assurer l'exclusion. Cette variable et les données globales associées ayant pu être modifiées par un processeur d'une autre molécule, il faut :

- effectuer le test and set en mémoire principale,
- invalider toutes les données globales du processus dans le cache, celles-ci risquant d'être périmées.

d) Demande d'invalidation des données d'un processus

Un processus peut être interrompu dans une molécule, par exemple pour des raisons d'entrées sorties et voir son exécution reprise dans une molécule différente. Si, après une deuxième interruption, il est repris par un processeur de la première molécule, il ne faut pas qu'il puisse accéder, dans le cache, à des données qui lui appartenaient mais qu'il a modifiées entre-temps. Aussi, dès qu'un processus cesse son activité il doit invalider toutes ses données (globales, locales).

Les invalidations de données globales (test and set) et des données d'un processus necessitent le balayage de la table NUMZ toute entière.
Ces opérations sont très longues : l'invalidation d'une ligne du cache prend un cycle d'horloge. Aussi, ces actions sont effectuées par morceau, de façon à ce que le cache ne soit pas bloqué pour les autres processeurs : on alterne l'invalidation d'une ligne avec la prise en considération d'une demande provenant d'un autre processeur (s'il y en a une).

e) Demande d'invalidation totale

Cette action est effectuée lors du démarrage du système, ou à la détection d'une panne d'un processeur de la molécule qui a pu perturber l'émission du préfixe d'identification. Elle provoque l'invalidation de tous les blocs du cache, et est effectuée intégralement, dès sa prise en compte. C'est une tâche longue (1 cycle d'horloge par ligne du cache) et qui bloque toute la molécule. Mais elle est extrêmement rare.

IV.3 DESCRIPTION DU MODELE

Le processeur cache a été simulé au niveau le plus fin en CASSANDRE.
Le lecteur trouvera dans [F7, rapport N° 1, Chapitre 4] un exposé du modèle détaillé. Pour la deuxième étape de simulation, qui nous interesse ici, les chemins de données entre le cache et la mémoire principale, et entre le cache et les processeurs ne sont pas décrits en tant que tels, dans un but d'optimisation des temps de simulation. Seuls sont modélisés les fils de contrôle et les connexions permettant de décrire les échanges à un niveau logique. Dans la même optique, la recherche d'une adresse émise par un processeur, par comparaison associative, est simulée par un appel de la procédure RANDOM.

Cette procédure répond OUI ou NON selon une probabilité qui est un paramètre de la simulation : OUI signifie que le mot demandé est présent dans le cache, NON indique qu'il faut d'abord aller chercher le bloc qui le contient en mémoire principale.

Le cache est sollicité de la part des processeurs par le tableau de signaux d'entrée DEM, à raison d'un vecteur par processeur. Pour chaque vecteur, l'un au plus des signaux est positionné à 1, avec la signification suivante :

DEM (0, I) : demande de lecture venant du processeur I
DEM (1, I) : demande d'écriture venant du processeur I
DEM (2, I) : demande de test and set venant du processeur I
DEM (3, I) : demande d'invalidation des données du processeur I
DEM (4, I) : demande d'invalidation totale venant du processeur I.

Le vecteur de signaux OK, à raison d'un fil par processeur, sert à véhiculer les messages d'acquittement des demandes :

OK(I) : acquittement du processeur I.

Pour les échanges avec la mémoire principale, on suppose que l'on dispose de deux tampons : l'un capable de contenir un bloc, l'autre contenant une file d'adresses et de données à écrire. Ainsi, il n'est pas nécessaire de synchroniser le cache et la mémoire principale pour qu'un échange ait lieu, et le bus mémoire n'est occupé que pendant un temps minimum. Comme nous l'avons vu au paragraphe II, les demandes d'échange se font par le signal ECHMEM, les réponses d'acquittement par le signal OKMEM.

Enfin, l'horloge HETAT, dont l'arrêt d'un processeur déclenche une impulsion, conditionne l'appel des deux procédures Fortran MOYEN et ETATC, qui impriment les mesures recueillies et l'état interne du processeur cache.

Il existe, à l'intérieur du processeur cache, une file d'attente de longueur égale au nombre de processeurs dans la molécule. Cette file est gérée en FIFO (premier arrivé, premier servi). Dans tous les cas où une demande ne peut pas être satisfaite immédiatement, soit parce qu'elle implique une invalidation partielle du cache, soit parce qu'elle necessite un échange avec la mémoire principale, elle est mise en file d'attente.

Le processeur laisse son fil positionné tant que la demande n'a pas été acquittée. Par contre, le cache masque les demandes qui sont mises en file d'attente, de façon à ne pas en tenir compte dans son processus de sélection.

Dans le modèle, la file d'attente est représentée par deux vecteurs d'entiers

FDM : file des numéros de processeurs
FDTYP : file des types de demandes.

La mise en file d'attente d'une demande d'écriture, par exemple, se traduit dans le programme par :

```
00001   'UNITE' PCACHE (H,HETAT,DEM(0:4,1:12),OKMEM(1:3);ECHMEM(1:3),OK(1:12));
00064   'HORLOGEMERE' H,HETAT;
00073   'TABLEAU' 'ENTIER'
00075               LFILE(0:12),     "LONGUEUR FILE D'ATTENTE"
00087               FDM(1:12),       "FILE DES DEMANDES MEMOIRE. ON Y MET LE NO
00097                                    DES PROCESSEURS"
00097             FDTYP(1:12);     "FILE DES TYPES DE DEMANDES MEMOIRE"

00211   'COMPTEUR' CIRCUL,      "DERNIERE DEMANDE PRISE EN COMPTE"
00219               INV,         "COMPTEUR DES LIGNES INVALIDEES"
00223           C1,C2,
00229               INACTIF,      "COMPTEUR CYCLES D'INACTIVITE"
00237           PTFDM;      "POINTEUR FILE DES DEMANDES MEMOIRE"

01595   ECRIRE:<H> 'P1'(PTFDM),
01614               FDM(PTFDM) $= CIRCUL, FDTYP(PTFDM) $=2,
01647               INHIBE(CIRCUL) <=0, 'ALLERA' SELECT1;
```

Après l'exécution d'un échange, d'une invalidation totale, ou de l'invalidation d'une ligne, le cache teste ses signaux d'entrées pour déterminer la prochaine action à entreprendre. Dans notre modèle, il y a deux points d'écoute privilégiés, qui sont les états SELECT et SELECT1, qui correspondent aux tests à priorité câblée suivants :

SELECT

- y a-t-il une demande venant d'un processeur non inhibé ?
- y a-t-il un échange en cours avec la mémoire principale ?
- y a-t-il au moins une demande en file d'attente ?

SELECT1

- y a-t-il un échange en cours avec la mémoire principale ?
- y a-t-il au moins une demande en file d'attente ?
- y a-t-il une demande venant d'un processeur non inhibé ?

Les processeurs ont une égale priorité.
Ils sont servis à tour de rôle, par balayage circulaire des demandes (utilisation du pointeur CIRCUL et des procédures SELECTE et SELECTE1).

Le résultat du test sert à indexer la variable d'état vectorielle ECLATM, qui est utilisée comme aiguillage.

```
00108   'REGISTRE' INHIBE(1:12),     "1-> LA DEMANDE DU PROC I PEUT ETRE PRISE"
00123              TBLOK(1:12,1:12),     "REPONSES POSITIVES A 1 PROC"
00140              ACCESM;      "1: ACCES MEMOIRE EN COURS"
00147   'ENTIER' NBP,       " NB DE PROCESSEURS"
00152            LMEM,     "NB DE LIGNES -1 DE MEM CACHE"
00157            AIG,      "VARIABLE DE CONTROLE AIGUILLAGE"
00161            BOOL,     "REPONSE PROCEDURE SELECTE1"
00166            DEMAND,      "RECUPERE LES DEMANDEURS, POUR 1 ACTION VERS CACHE"
00173            REPMEM,      "ENREGISTREMENT D' 1 REPONSE MEMOIRE"
00180            LBLOC, TRAV,      "VARIABLES DE TRAVAIL"
00191            TPSLEC,      "TEMPS -2 DE LECTURE D'1 BLOC"
00198            RAND,ICIBLOC;      "POUR GENERATION NB ALEATOIRE"

00243   'ETAT' ECLATM(0:9),  "NOREP,REPLEC,REPECR,REPTES"
00256                  "CALLMEM,LIRE,ECRIRE,TES,INVAL,RAZ"
00256          ETINV(3:4);   "FININV,ASKTES"
00267   'FORTRAN' DECALE(3),ETATC(9),MOYEN(1);
00296   'PROCEDURE' SELECTE(3),SELECTE1(4),RANDOM(2);

01368   SELECT1: <H> 'SI' ACCESM 'ALORS' AIG $= REPMEM
01398                'SINON'
01399               ('SIE' PTFDM>0 'ALORS' AIG $= 4
01414                'SINON'
01415                 (DEMAND $= $(INHIBE.(/+DEM)),
01441                  'SIE' DEMAND = 0 'ALORS' AIG $= 0
01466                  'SINON'
01457                    ('CALL' SELECTE1(CIRCUL,NBP,DEMAND,BOOL),
01492                     'SIE' BOOL=0 'ALORS' AIG $= 0  'SINON'
01506                      ('SI' DEM(,CIRCUL) 'ALORS'
01521                           (AIG $= 5)
01528                           (AIG $= 6)
01535                           (AIG $= 7)
01542                           (AIG $= 8)
01549                           (AIG $= 9, INV $= LMEM )))),
01570            TRAV $= FDM(1);
01582            'FAIRE' ECLATM(AIG);
```

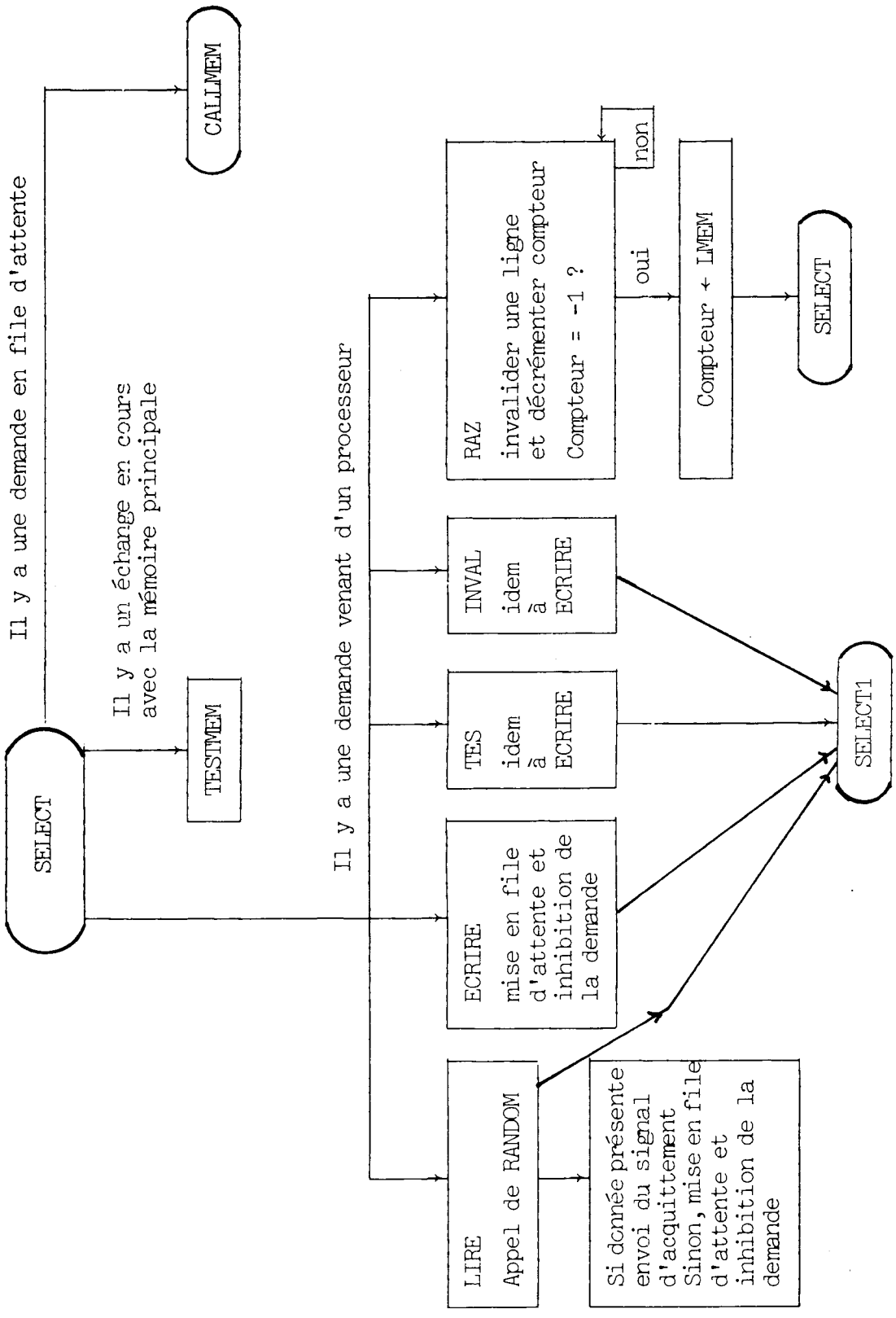
Les paramètres pour la simulation du cache sont :

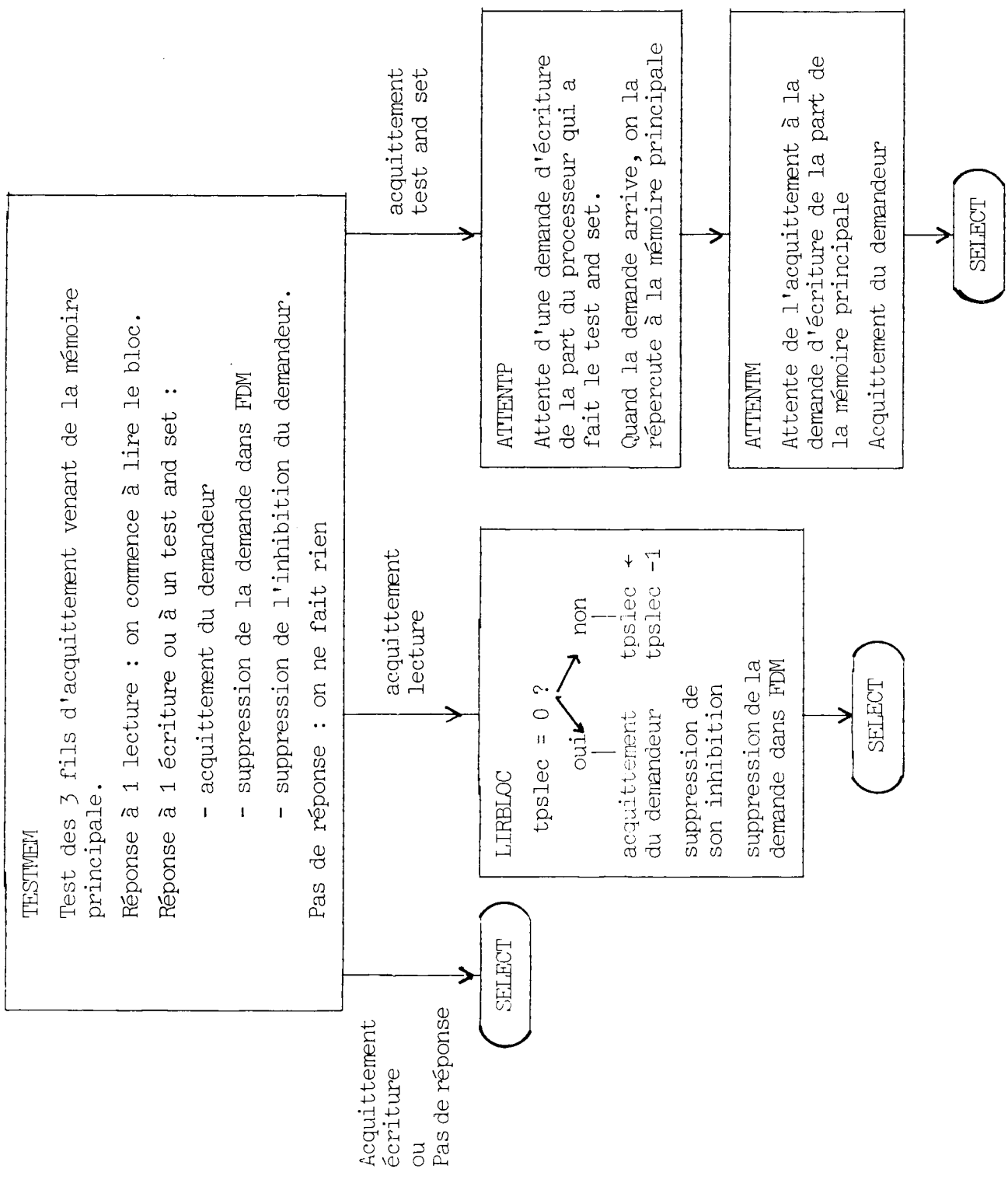
- le nombre de lignes LMEM, qui influe sur le temps d'invalidation
- la taille du bloc, qui détermine le temps TPSLEC nécessaire pour le ranger du tampon en mémoire locale
- le taux d'échec en lecture, nécessaire à la procédure RANDOM.

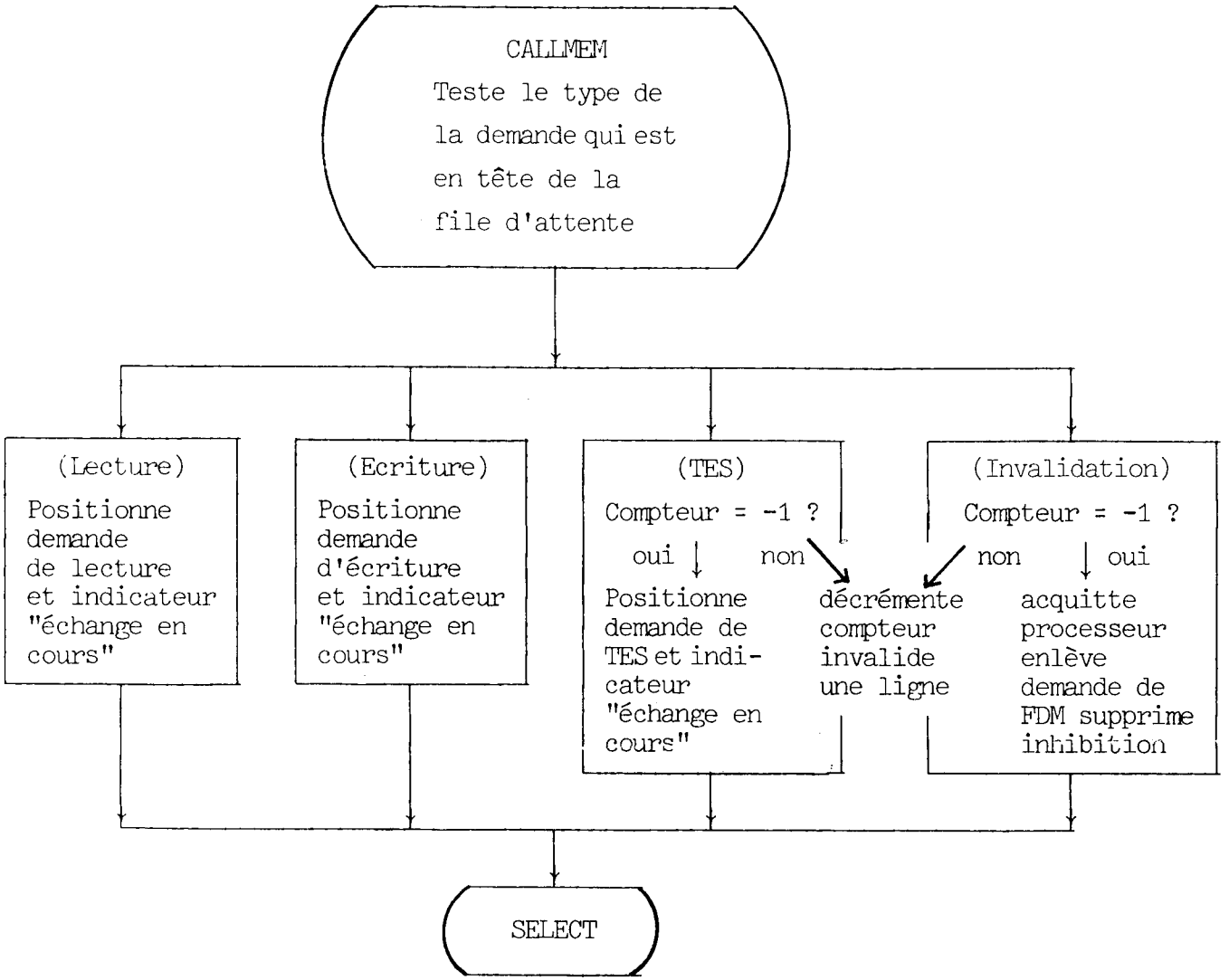
Les mesures recueillies dans l'unité PCACHE sont le nombre de cycles d'inactivité (INACTIF), et l'histogramme de la longueur de la file d'attente des demandes non satisfaites (LFILE).

Le programme correspondant à l'unité PCACHE est donné en annexe III. C'est la représentation, dans le langage LASCAR, de l'algorithme de gestion que détaillent les trois pages ci-après.

IV.4 ALGORITHME DE GESTION DU CACHE




TESTMEM
Test des 3 fils d'acquittement venant de la mémoire principale.
Réponse à 1 lecture : on commence à lire le bloc.
Réponse à 1 écriture ou à un test and set :
- acquittement du demandeur
- suppression de la demande dans FDM
- suppression de l'inhibition du demandeur.
Pas de réponse : on ne fait rien
Acquittement écriture ou Pas de réponse
SELECT
acquittement lecture
LIRBLOC
tpslec = 0 ?
oui
non
acquittement du demandeur
tpslec ← tpslec -1
suppression de son inhibition
suppression de la demande dans FDM
SELECT
acquittement test and set
ATTENTP
Attente d'une demande d'écriture de la part du processeur qui a fait le test and set.
Quand la demande arrive, on la répercute à la mémoire principale
ATTENTM
Attente de l'acquittement à la demande d'écriture de la part de la mémoire principale
Acquittement du demandeur
SELECT


CALLMEM
Teste le type de la demande qui est en tête de la file d'attente
(Lecture)
Positionne demande de lecture et indicateur "échange en cours"
(Ecriture)
Positionne demande d'écriture et indicateur "échange en cours"
(TES)
Compteur = -1 ?
oui
non
Positionne demande de TES et indicateur "échange en cours"
décrémente compteur invalide une ligne
(Invalidation)
Compteur = -1 ?
non
oui
acquitte processeur enlève demande de FDM supprime inhibition
SELECT

V . LA MÉMOIRE PRINCIPALE : PMEM

Le modèle de la mémoire principale est celui qui est le plus éloigné d'un modèle de description de matériel. En effet, compte tenu du type de simulation envisagé, deux hypothèses simplificatrices ont permis d'alléger l'unité PMEM :

- Nous ne simulons qu'une molécule, en négligeant, à ce premier stade, l'influence qu'aurait la présence d'autres sous-systèmes. Une gestion de multiples demandes à la mémoire principale n'est donc pas nécessaire.

- La mémoire principale est paginée. Cependant, le temps de transmission d'une page étant de l'ordre de la dizaine de milli-secondes, et l'unité de temps simulée étant de l'ordre de la centaine de nano-secondes, ces deux grandeurs sont de nature trop différente pour être intégrées dans une même simulation. Le phénomène de défaut de page n'est donc, à ce niveau de simulation, pas pris en considération.

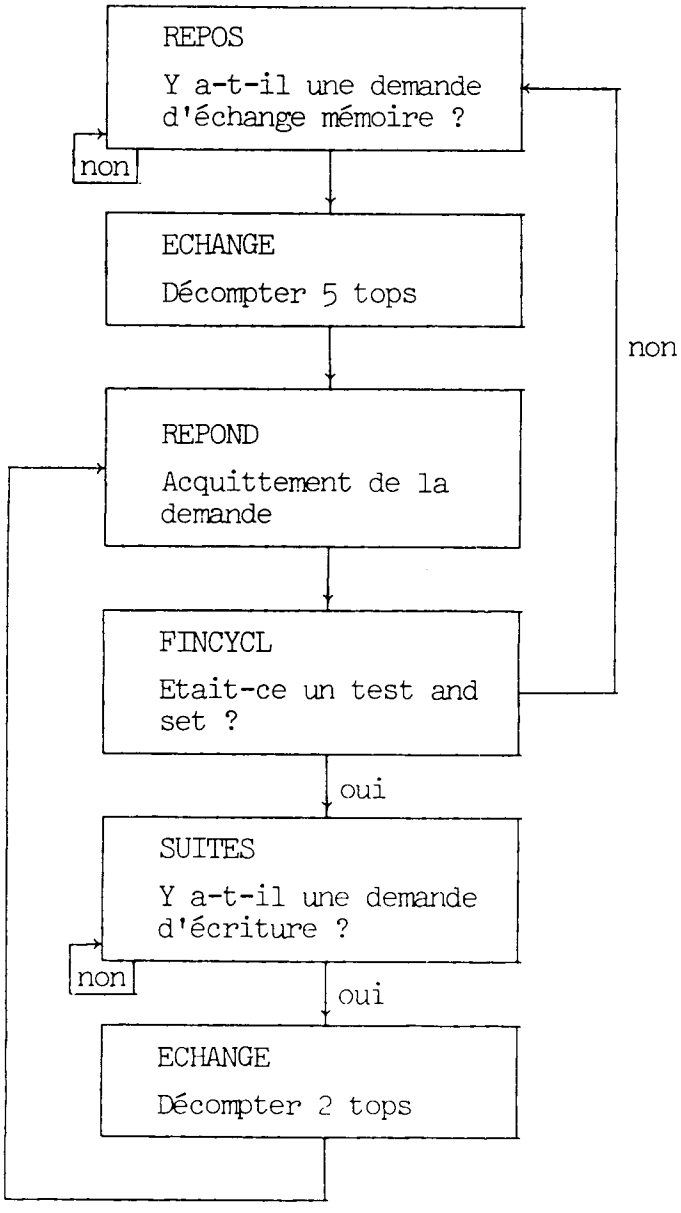
L'écriture d'un modèle de mémoire principale plus complet, tenant compte à la fois des fautes de page et des conflits d'accès de la part de plusieurs molécules est toutefois immédiate : il suffit de prévoir les signaux de connexion vers plusieurs molécules. Les procédures de sélection d'un demandeur, et de génération de réponses aléatoires avec une probabilité donnée, seraient identiques à celles qui ont été écrites pour le processeur cache.

Dans l'exemple que nous présentons, nous avons supposé que le temps de cycle de la mémoire principale est de 1280 nano-secondes, son temps de réponse 960 nano-secondes, et que la largeur du bus mémoire est suffisante pour véhiculer un bloc de mémoire cache en parallèle. Ces chiffres ne sont que des paramètres, et en changer la valeur revient à modifier les tests d'égalité qui sont faits sur le compteur A.

Les mesures recueillies dans l'unité PMEM sont le nombre de lectures, le nombre d'écritures, le nombre de test and set effectués pendant la simulation, ainsi que le temps d'inactivité de PMEM.

Le sous-programme Fortran ETATM, qui ne s'exécute que lorsque, un processeur s'arrêtant, l'unité MODELE envoie l'impulsion HETAT, permet d'obtenir les valeurs des compteurs de prise de mesures.

Algorithme



Description LASCAR

```
U. S.   TEXTE SOURCE

00001   'UNITE' PHLM(H,HETAT,ECHMEM(1:3); OKMEM(1:3));
00039   'HORLOGEMERE' H,HETAT;
00048   'COMPTEUR' KLEC,KECR,KTES,
00064             "COMPTE LE NB DE LECTURES,ECRITURES,TEST AND SET"
00064               A,       "POUR INTRODUIRE DES RETARDS"
00066               TYPE,    " 1:LECTURE, 2:ECRITURE, 3:TEST AND SET"
00071              ATTEND;  "COMPTE LES CYCLES D'INACTIVITE"
00078   'FORTRAN' ETATM(4);
00088   <HETAT>  'CALL' ETATM(KLEC,KECR,KTES,ATTEND);
00125   REPOS: <H> 'SI' ECHMEM 'ALORS'
00142                (TYPE $=1, 'P1'(KLEC), 'ALLERA' ECHANGE)
00167                (TYPE $=2, 'P1'(KECR), 'ALLERA' ECHANGE)
00192                (TYPE $=3, 'P1'(KTES), 'ALLERA' ECHANGE),
00218             'SIE' TYPE=0 'ALORS' 'P1'(ATTEND);
00236   ECHANGE: <H> 'P1'(A),
00252                'SIE' A=5 'ALORS' 'ALLERA' REPOND;
00265   REPOND: OKMEM(TYPE) :=1;
00286        <H> 'ALLERA' *;
00292   FINCYCL: OKMEM(TYPE):=0;
00314        <H> A $= 0,
00321            'SIE' TYPE = 3 'ALORS' 'ALLERA' SUITES
00336            'SINON' 'ALLERA' REPOS,
00344            TYPE $= 0;
00351   SUITES: <H> 'SI' ECHMEM(2) 'ALORS'
00372                  (TYPE $= 2, A $= 3, 'ALLERA' ECHANGE);
00394
```

VI . PREMIERS RESULTATS

Le modèle qui vient d'être présenté a été simulé avec 2, 4 et 8 processeurs. La même trace a été utilisée pour les trois simulations. Dans le cas où l'on a 8 processeurs, chaque processeur travaille sur 1/8 de la trace totale, et est par exemple simulé sur une trace 4 fois plus petite que dans le cas où l'on a 2 processeurs.

VI.1 CARACTERISTIQUES DE LA TRACE, ET VALIDATION

La trace totale contient 4098 instructions. La routine de décodage en a écarté 58, pour lesquelles nous n'avions pas de signature, ce qui représente un pourcentage de 1,4 %. Il est certain que l'erreur relative commise sur les temps d'exécution, en ne simulant pas ces instructions, est supérieure à 1,4 % ; mais, en gardant à l'esprit que les résultats que nous donnons sont optimistes, nous pouvons affirmer que l'ordre de grandeur est respecté.

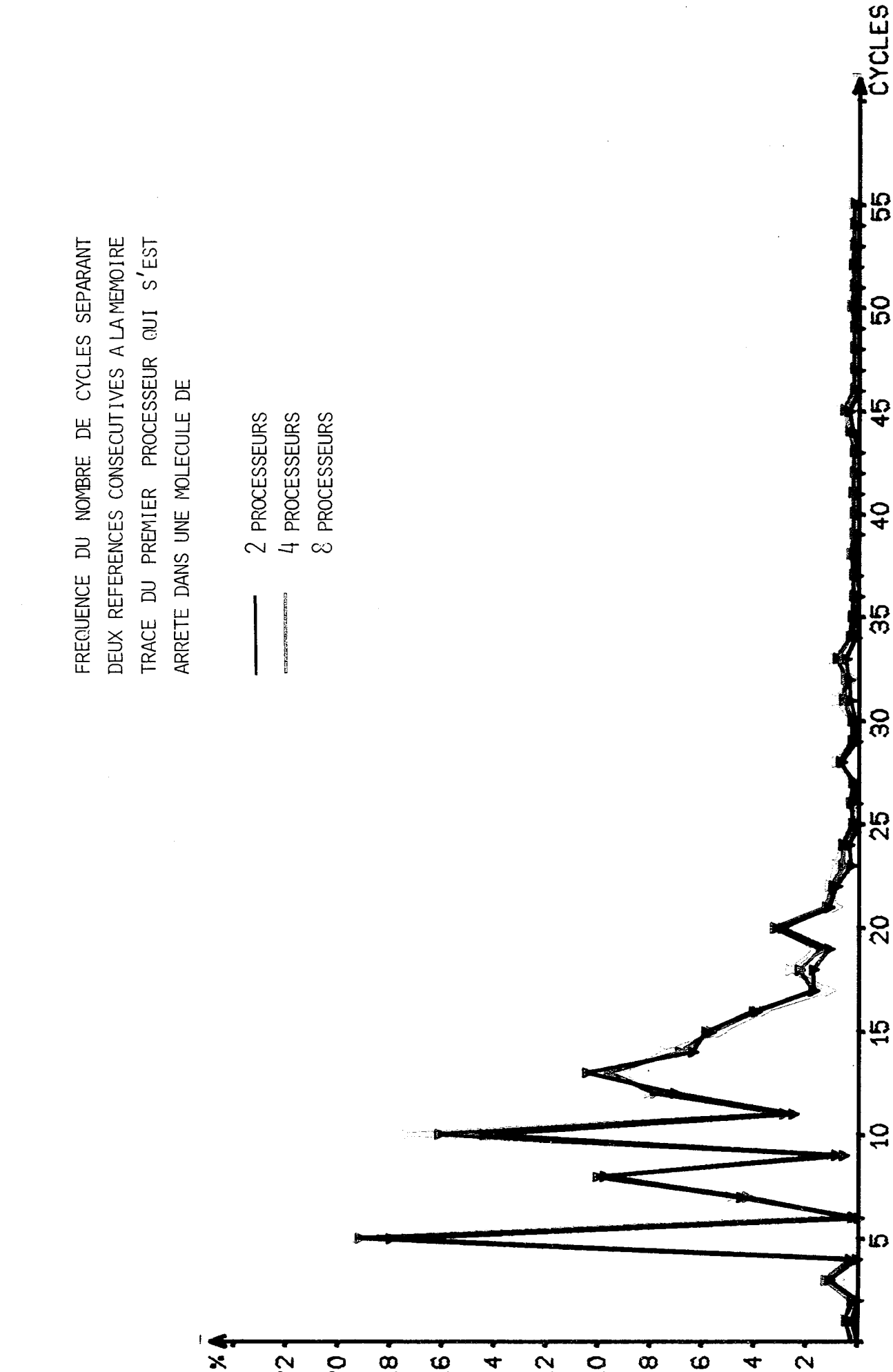
Chaque instruction étant connue par sa signature, la trace exécutée est caractérisée par l'histogramme des temps séparant deux références consécutives à la mémoire. Ces histogrammes ont été relevés, pour chaque simulation, et pour chaque sous-trace de chaque processeur. Les temps sont comptés en nombre de cycles d'horloge ; et, caractérisant uniquement la trace d'instructions, ces temps ne tiennent pas compte des attentes éventuelles sur accès au cache. Par contre, la séquence d'adressage et de chargement de l'instruction est comptée.

La figure 1 de la page suivante représente ces histogrammes, ramenés à des fréquences, pour la trace exécutée par le premier processeur qui s'est arrêté, sur les 2, 4 et 8 processeurs de la molécule simulée. Ces trois courbes, sans être parfaitement superposables, sont tout à fait semblables. Ce qui montre que, pour le jeu d'essai utilisé pour notre simulation, un huitième de la trace totale, c'est-à-dire la sous-trace exécutée par un processeur, lorsqu'il y en a 8, est représentative de la trace toute entière. En d'autres termes, 500 instructions sont suffisantes pour représenter le comportement de la trace totale (nous avons vérifié que ces instructions n'étaient pas enfermées dans des boucles).

Les temps donnés en abscisse sur la figure 1 s'arrêtent à 56 cycles.
Il existe cependant quelques points isolés, pour lesquels l'abscisse est supérieure à 60. Leur fréquence est de 0,6 %. Ces points correspondent aux instructions de multiplication et de division, ainsi qu'à l'instruction d'invalidation partielle générée en fin de trace.

- Figure 1 -

FREQUENCE DU NOMBRE DE CYCLES SEPARANT DEUX REFERENCES CONSECUTIVES A LA MEMOIRE TRACE DU PREMIER PROCESSEUR QUI S'EST ARRETE DANS UNE MOLECULE DE



Les autres caractéristiques de la trace sont les suivantes :

- nombre de lecture par instruction : 1, 3
- nombre d'écriture par instruction : 0, 2
- nombre de test and set pour 100 instructions : 0, 4
- temps moyen entre deux références consécutives à la mémoire : 14,5 cycles.

Les instructions se répartissent dans les différents types (cf annexe II) selon le tableau suivant :

TYPE	NOMBRE D'INSTRUCTIONS
0	501
1	1580
2	1196
3	638
4	10
5	56
6	16
7	23
8	18
9	0
10	1 par processeur

VI.2 PARAMETRES DE LA SIMULATION

Les résultats de simulation que nous présentons ici dépendent des valeurs suivantes des paramètres du modèle :

- Période de l'horloge mère : 160 nano-secondes
- Temps d'accès à la mémoire principale : 960 nano-secondes
- Temps de cycle de la mémoire principale : 1280 nano-secondes
- Taille du cache : 256 lignes
  4 colonnes
- Taille du bloc : 16 octets
- Taux d'échec du cache, en lecture : 10 %
  Ce taux d'échec provient des résultats d'une simulation moins détaillée réalisée par M. MAZARE pour la même taille de cache et de bloc, sur une trace volumineuse dont nous avons extrait notre trace [F7, rapport n°3, Chapitre 2].

VI.3 RESULTATS A L'ARRET DU PREMIER PROCESSEUR

Les résultats donnés dans ce paragraphe caractérisent le comportement de la molécule à pleine charge. Le tableau 1 résume les mesures recueillies à l'arrêt du premier processeur arrivé en fin de trace, et permet de faire une comparaison aisée de performances en fonction du nombre de processeurs.

Sur la trace que nous avons sélectionnée, et qui, rappelons-le, ne contient aucune instruction décimale, aucune instruction en virgule flottante ni aucune instruction de mémoire à mémoire, un processeur unique relié au cache est capable d'exécuter 236.10^3 instructions par seconde.

Lorsque l'on connecte deux processeurs au cache, la puissance de l'ensemble est de 451.10^3 instructions par seconde, soit un facteur 1,91 par rapport à la puissance d'un seul processeur.

Lorsque l'on passe de 2 à 4 processeurs, la puissance de la molécule est multipliée par 1,77. Lorsque l'on passe de 4 à 8 processeurs, la puissance de la molécule n'est plus multipliée que par 1,64. Le premier doublement du nombre de processeurs est donc nettement plus intéressant que le second.

Les chiffres que nous donnons ici représentent uniquement le débit maximal de la molécule à plein rendement, et ne tiennent compte ni du temps de chargement des contextes dans les processeurs, ni de "l'overhead" logiciel de gestion et de synchronisation de tâches parallèles.

Le taux d'inactivité du cache est inversement proportionnel au nombre de processeurs dans la molécule. Lorsque la molécule contient 2 processeurs, le cache est inactif plus de la moitié du temps. Pour 4 processeurs, la proportion du temps d'inactivité tombe à 1/4. Pour 8 processeurs, le cache est pratiquement saturé, puisque le pourcentage de temps d'inactivité est à peine supérieur à 3 %. Il ne faut donc pas envisager de faire tourner simultanément plus de 8 processeurs par molécule. C'est pourquoi nous n'avons pas simulé une molécule contenant 12 processeurs, comme cela avait été prévu au départ.

Le taux d'inactivité de la mémoire principale, quoique dans une moindre mesure que celui du cache, est également très sensible au nombre de processeurs par molécule. Ceci est dû en particulier à l'hypothèse d'écriture immédiate en mémoire principale qui a été faite.

Regardons maintenant plus en détail certains aspects des mesures recueillies.

TABLEAU 1

ARRET DU PREMIER PROCESSEUR	2 processeurs	4 processeurs	8 processeurs
Molécule			
. nombre de cycles simulés	55 771	29 907	17 838
. nombre d'instructions exécutées par la molécule	4 025	3 835	3 761
. puissance de la molécule en nombre d'instructions par seconde	451.10^3	801.10^3	1317.10^3
. nombre moyen de cycles par instruction	13,85	7,79	4,74
Mémoire principale			
. nombre de lectures	560	531	453
. nombre d'écritures	858	829	814
. nombre de test and set	16	16	10
. % temps d'inactivité	79,29 %	62,94 %	42,48 %
Cache			
. longueur moyenne de FDM	0,38	1,11	2,91
. % temps d'inactivité	53,08 %	24,46 %	3,49 %
Premier processeur arrêté			
. nombre d'instructions exécutées	2 019	1 010	507
. puissance du processeur en nombre d'instructions par seconde	226.10^3	211.10^3	177.10^3
. attente moyenne en lecture (cycles)	1,49	2,89	5,42
. attente moyenne en écriture (cycles)	9,93	16,74	28,65
. attente moyenne sur test and set (cycles)	267	280	320
. % temps d'attente	19,28 %	28,04 %	42,54 %
. nombre moyen de cycles par instruction	27,62	29,61	35,18

a) Attente en lecture

Pour les trois simulations, nous avons relevé l'histogramme des temps d'attente, exprimés en nombres de cycles d'horloge, entre le moment où le processeur est prêt à recevoir la donnée et le moment où le signal d'acquittement est reçu. Une attente nulle signifie que la donnée est disponible deux tops d'horloge après le positionnement de la demande, donc récupérable par la micro-instruction suivante sans que le processeur soit ralenti. La figure 2 de la page suivante donne les courbes de fréquence de ces attentes pour 2, 4 ou 8 processeurs.

Sur la courbe noire, qui représente la fréquence des temps d'attente du premier processeur qui s'est arrêté dans une molécule contenant deux processeurs, 84 % des lectures sont sans attente, et 89 % provoquent une attente de moins de deux cycles. Le deuxième pic, qui correspond à 10 % des lectures, se situe à 9 et 10 cycles : ce sont les lectures de données qui ne sont pas dans le cache, et qu'il faut aller chercher en mémoire principale. Le nombre d'attentes supérieures à 10 cycles est négligeable. Il n'y a donc pratiquement aucune gêne due à la présence d'un deuxième processeur.

Lorsque 4 processeurs sont présents dans la molécule (courbe bleue), un peu moins de 74 % des lectures sont sans attente. Les autres lectures de données présentes dans le cache peuvent provoquer de 1 à 5 cycles d'attente. On retrouve le pic à 9 et 10 cycles, mais moins prononcé, et s'étalant jusqu'à 20 cycles d'attente pour les données qui ne sont pas dans le cache.
La présence des 3 autres processeurs est donc sensible, mais le taux de conflits est acceptable.

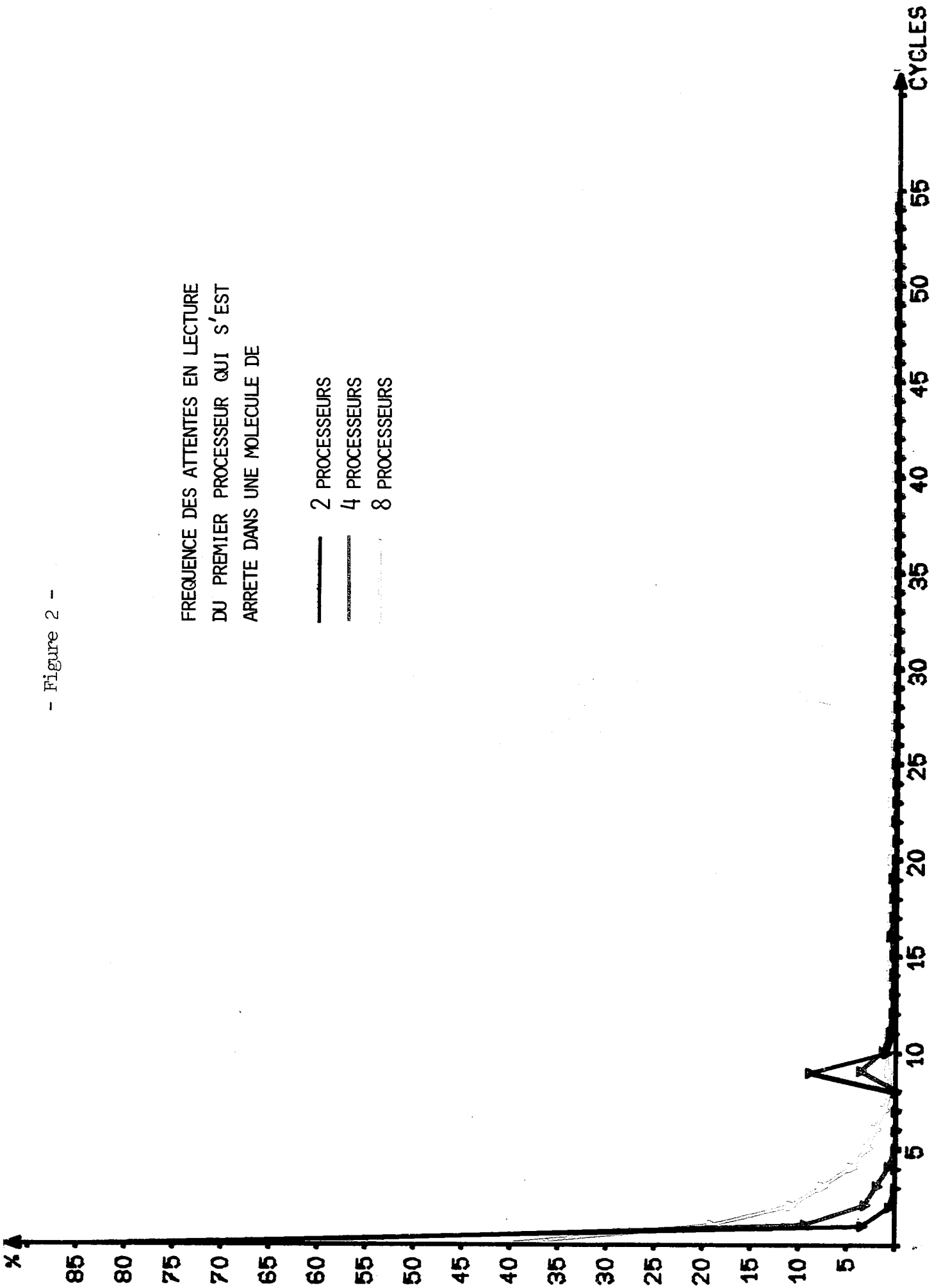
Pour une molécule de 8 processeurs (courbe rouge), seulement 44 % des lectures ne provoquent pas d'attente. Il peut falloir plus de 8 cycles pour accèder à une donnée présente dans le cache. Le pic correspondant aux données absentes a totalement disparu, l'attente sur ces données s'étageant uniformément entre 9 et 25 cycles. La présence des 7 autres processeurs est très pénalisante pour l'accès au cache en lecture.

Il existe, pour les trois courbes que nous venons d'analyser, quelques points isolés qui représentent une attente très longue. La signification de ces points isolés est la même que celle des points isolés pour les courbes d'attente en écriture ; nous la verrons plus loin.

- Figure 2 -

FREQUENCE DES ATTENTES EN LECTURE
DU PREMIER PROCESSEUR QUI S'EST
ARRETE DANS UNE MOLECULE DE

2 PROCESSEURS
4 PROCESSEURS
8 PROCESSEURS



b) Attente en écriture

L'écriture d'une donnée se faisant toujours en mémoire principale, il ne peut pas y avoir d'attente nulle sur écriture. En l'absence de tout conflit, le temps minimum d'écriture, compte tenu de l'accès au cache d'abord, à la mémoire principale ensuite, est de 8 cycles d'horloge. Seule exception, la ré-écriture d'une donnée sur laquelle on vient de faire un test and set ne prend que 2 cycles d'attente. Ce temps de ré-écriture est indépendant du nombre de processeurs, car pendant l'exécution d'un test and set, tous les accès au cache et à la mémoire principale sont bloqués. Dans la suite, nous ne reparlerons donc plus de ces écritures très particulières.

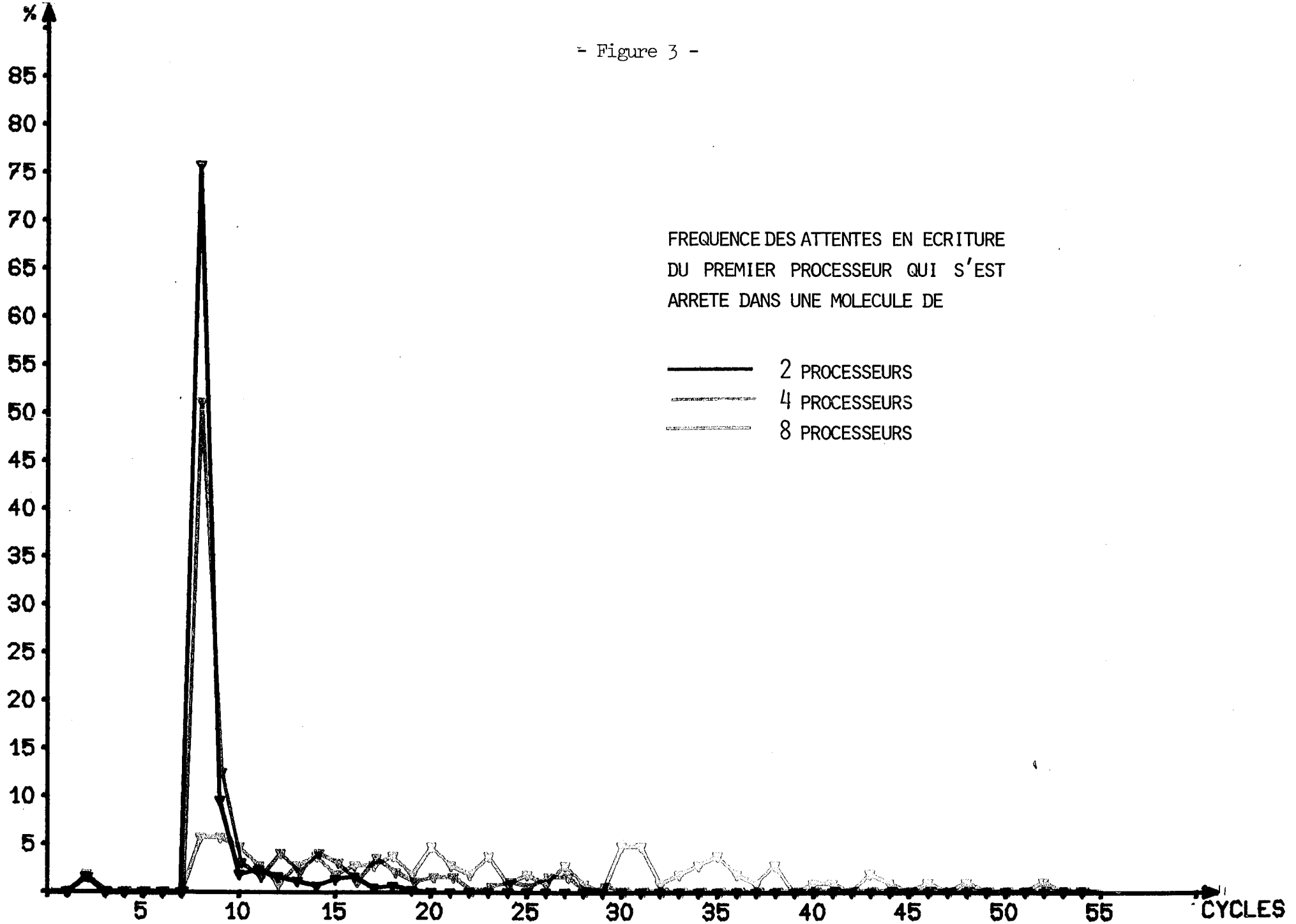
La figure 3 donne les courbes de fréquence des attentes sur écritures, pour 2, 4 et 8 processeurs.

Pour deux processeurs (courbe noire), le taux d'attentes minimales est de 76 %. Les attentes de 8 à 10 cycles représentent un pourcentage de 87 %. Les 11 % restant se situent entre 11 et 20 cycles, et représentent les écritures qui sont en file d'attente derrière une demande d'écriture de l'autre processeur. Leur pourcentage est donc faible, et les attentes ne dépassent pas un seuil de 20 cycles.

Lorsque la molécule contient quatre processeurs (courbe bleue), le taux d'attentes minimales n'est plus que de 51 %. Un peu plus de 23 % des attentes se situent entre 11 et 20 cycles, et les attentes de 20 à 30 cycles représentent un pourcentage de 7 %, ce qui n'est pas négligeable compte tenu de la longueur de l'attente. L'influence des 3 autres processeurs est donc nettement sensible.

Sur la courbe rouge, qui illustre une molécule de 8 processeurs, les attentes minimales sont réduites. Les attentes sont uniformément réparties entre 8 et 38 cycles d'horloge. Entre 8 et 10 cycles, on ne trouve que 17 % des écritures. De 11 à 20 cycles, le pourcentage est de 30 % ; de 21 à 30 cycles, il est de 22 % ; de 31 à 40 cycles, il est également de 22 %. Les 7 % restant se situent entre 41 et 55 cycles. Lorsqu'il y a 8 processeurs dans la molécule, les attentes en écriture deviennent prohibitives.

- Figure 3 -


%
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
5
10
15
20
25
30
35
40
45
50
55
CYCLES
FREQUENCE DES ATTENTES EN ECRITURE
DU PREMIER PROCESSEUR QUI S'EST
ARRETE DANS UNE MOLECULE DE
2 PROCESSEURS
4 PROCESSEURS
8 PROCESSEURS

En résumé, des conclusions analogues peuvent être tirées pour les ralentissements imposés aux processeurs par les échanges avec la mémoire, qu'il s'agisse de lectures ou d'écritures. Lorsque la molécule contient deux processeurs, l'influence réciproque d'un processeur sur l'autre est négligeable. Pour une molécule de quatre processeurs, les conflits d'accès au cache sont plus fréquents, et chaque processeur subit un léger ralentissement dû à la présence des trois autres. Lorsque huit processeurs sont connectés au même cache, les temps d'attente dûs aux échanges avec la mémoire deviennent très pénalisants. Il suffit pour s'en convaincre de se reporter aux résultats du Tableau I. Le passage de 2 à 4 processeurs provoque une baisse de performance du processeur de 6,6 %, ce qui est peu. Par contre, un processeur qui se trouve dans une molécule de 8 processeurs subit, par rapport à celui d'une molécule de 4 processeurs, une perte de puissance de 16 %. Huit processeurs exécutant simultanément des programmes est probablement un chiffre déjà trop élevé, et sans aucun doute une borne supérieure à ne pas dépasser.

Sur la dernière figure que nous venons d'analyser, n'apparaissent pas quelques points isolés, très peu nombreux, mais qui représentent une attente de plusieurs centaines de cycles. Ces points correspondent aux écritures bloquées en file d'attente derrière un test and set. En effet, un test and set provoque l'invalidation de toutes les données globales d'un processus, ce qui prend un cycle par ligne du cache, sans compter les cycles utilisés entre-temps à servir d'autres processeurs en lecture.

Un test and set peut donc bloquer pendant un temps très long plusieurs processeurs, ce qui n'est pas tolérable. Il faudra donc revenir sur la conception du processeur cache, en introduisant une deuxième file d'attente, réservée au test and set et aux invalidations.

c) File d'attente des demandes

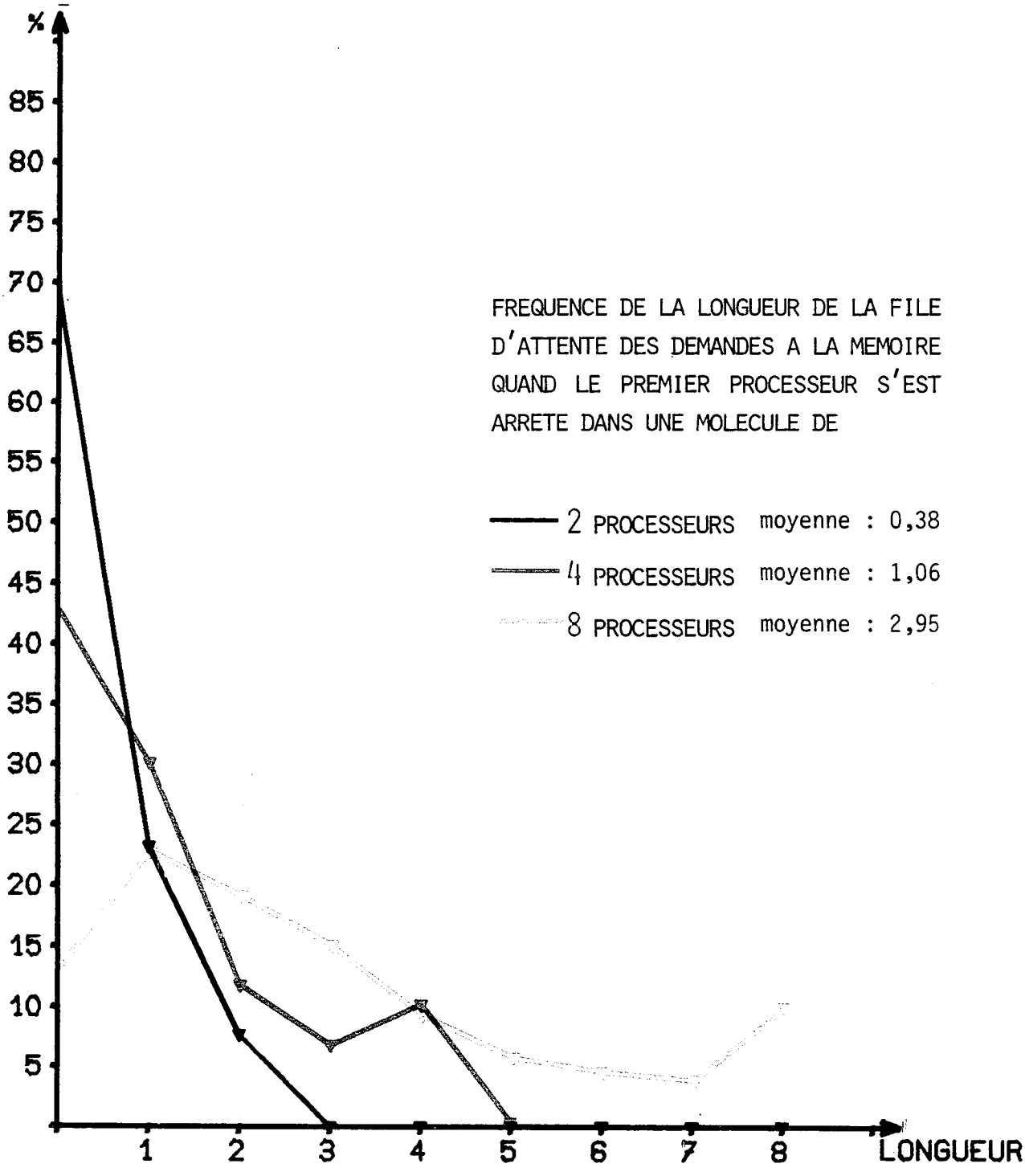
Pour les trois simulations, l'histogramme de la longueur de la file d'attente FDM a été établi. Ramenées à des fréquences, les trois courbes sont représentées sur la figure 4. La longueur maximale de FDM est évidemment égale au nombre de processeurs de la molécule.

Lorsqu'il y a 2 processeurs, FDM est vide 69 % du temps. Lorsqu'il y en a 4, FDM contient 0 ou 1 élément 73 % du temps. Par contre, pour 8 processeurs, FDM contient 2 éléments ou plus 65 % du temps.

On peut noter que, pour la courbe correspondant à 4 (respectivement 8) processeurs la file d'attente contient 4 (respectivement 8) éléments plus souvent que 3 (respectivement 7) éléments.

Ce phénomène peut s'expliquer par le fait que, lors d'un test and set, FDM se remplit, et tous les processeurs peuvent voir leur demande bloquée en file d'attente pendant un temps assez long. Dès que le test and set est acquitté, la file se vide très vite, les autres demandes étant selon toute probabilité des lectures et des écritures.

Cette analyse nous confirme dans notre opinion sur la nécessité de séparer, dans le cache, les demandes courtes (lectures, écritures) des demandes longues (test and set, invalidation des données d'un processus), et de créer pour ce faire une deuxième file d'attente.


%
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
1
2
3
4
5
6
7
8
LONGUEUR
FREQUENCE DE LA LONGUEUR DE LA FILE D'ATTENTE DES DEMANDES A LA MEMOIRE QUAND LE PREMIER PROCESSEUR S'EST ARRETE DANS UNE MOLECULE DE
2 PROCESSEURS moyenne : 0,38
4 PROCESSEURS moyenne : 1,06
8 PROCESSEURS moyenne : 2,95

- Figure 4 -

VI.4 RESULTATS A L'ARRET DU DERNIER PROCESSEUR

Réunir l'ensemble des mesures, lorsque tous les processeurs sont arrêtés, revient à considérer le temps d'exécution apparent d'une trace dont le parallélisme est supposé parfait (toutes les branches contiennent le même nombre d'instructions), compte non tenu de l'exécution des primitives système (FORK et JOIN par exemple) de création de tâche et de synchronisation. En effet, dans cette simulation, les processeurs, une fois arrivés en fin de trace, ne sont pas redémarrés. L'ensemble des résultats est regroupé dans le tableau 2.

Si l'on compare les tableaux 1 et 2, on note peu de différence en ce qui concerne les chiffres relatifs au cache. Pour la mémoire principale, les résultats sont similaires pour les simulations à 2 et 4 processeurs ; par contre, le taux d'inactivité augmente fortement pour 8 processeurs, ce qui s'explique par le fait que, dans l'intervalle de temps nécessaire aux 7 invalidations partielles lorsque les processeurs arrivent en fin de trace, aucun échange n'a lieu avec la mémoire principale.

On note, dans le tableau 2, une baisse de puissance de la molécule. En effet, vers la fin de la simulation, certains processeurs sont arrêtés. Les chiffres du tableau 2 sont sans doute plus réalistes en moyenne, car on peut supposer qu'en fonctionnement normal, les processeurs ne seront pas constamment tous actifs.

Il est intéressant de comparer les temps d'exécution de la trace sur les trois molécules simulées.

Lorsque l'on passe de 2 à 4 processeurs, le temps d'exécution de la trace est divisé par 1,69. Lorsque l'on passe de 4 à 8 processeurs, ce temps est divisé par 1,47. Là encore, on peut noter que le premier doublement du nombre de processeurs est beaucoup plus intéressant que le second.

TABLEAU 2

ARRET DU DERNIER PROCESSEUR	2 processeurs	4 processeurs	8 processeurs
Molécule			
. nombre de cycles simulés	56 310	33 330	22 645
. nombre d'instructions exécutées par la molécule	4 040	4 041	4 045
. puissance de la molécule en nombre d'instructions par seconde	448.10^3	757.10^3	1116.10^3
. nombre moyen de cycles par instruction	13,93	8,24	5,59
Mémoire principale			
. nombre de lectures	564	562	497
. nombre d'écritures	860	860	860
. nombre de test and set	16	16	16
. % temps d'inactivité	79,40 %	65,24 %	51,14 %
Cache			
. longueur moyenne de FDM	0,38	1,06	2,95
. % temps d'inactivité	52,91 %	26,95 %	3,69 %
Dernier processeur arrêté			
. nombre d'instructions exécutées	2 021	1 010	506
. puissance du processeur en nombre d'instructions par seconde	223.10^3	189.10^3	139.10^3
. attente moyenne en lecture (cycles)	1,21	2,86	6,50
. attente moyenne en écriture	10,57	19,96	46,75
. attente moyenne sur test and set	297	316	300
. % temps d'attente	19,30 %	29,44 %	47,30 %
. nombre moyen de cycles par instruction	27,86	33,00	44,75

VII . SIMULATION DE LA MOLÉCULE AVEC UN CACHE MODIFIÉ

Nous avons vu au paragraphe précédent, en analysant les mesures prises au cours des premières simulations, qu'un processeur était capable de paralyser l'activité de la molécule lorsqu'il exécutait un test and set. Nous avons émis l'hypothèse que l'algorithme de gestion des demandes du cache pouvait en être responsable. Nous avons donc recommencé une simulation de la molécule, avec un cache modifié, sur la même trace d'instructions dans le cas où ce phénomène était le plus pénalisant, c'est-à-dire pour 8 processeurs.

VII.1 ALGORITHME DU CACHE MODIFIE

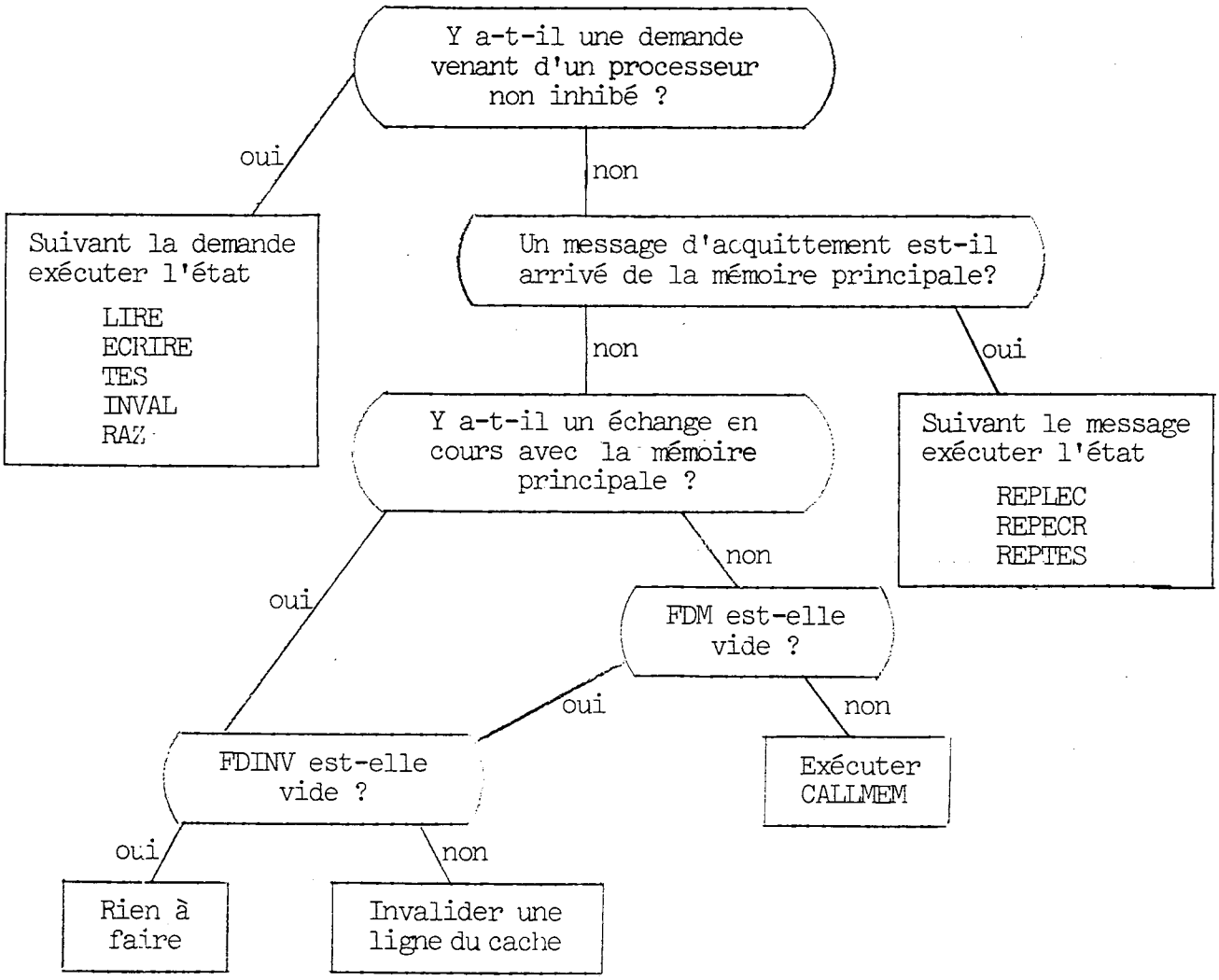
Le cache est maintenant doté de deux files d'attentes, où sont rangées les demandes des processeurs qui ne peuvent pas être immédiatement satisfaites :

- FDM contient les demandes qui nécessitent un échange avec la mémoire principale

- FDINV contient les demandes qui impliquent une invalidation des données contenues dans le cache.

La file FDM est plus prioritaire que la file FDINV. Les deux points d'écoute, SELECT et SELECT1, ont donc été modifiés pour tenir compte de ce nouvel élément de décision, en vue du choix de la prochaine action à entreprendre. Les tests ont été rendus plus complexes, du fait qu'il est maintenant possible que l'on soit simultanément en attente d'un message de la mémoire principale et en cours d'invalidation.

SELECT


Y a-t-il une demande venant d'un processeur non inhibé ?
oui
non
Suivant la demande exécuter l'état
LIRE
ECRIRE
TES
INVAL
RAZ
Un message d'acquittement est-il arrivé de la mémoire principale?
non
oui
Suivant le message exécuter l'état
REPLEC
REPECR
REPTES
Y a-t-il un échange en cours avec la mémoire principale ?
oui
non
FDM est-elle vide ?
oui
non
FDINV est-elle vide ?
Exécuter CALLMEM
oui
non
Rien à faire
Invalider une ligne du cache

SELECT1 est analogue à SELECT dans son principe, mais le test d'une demande venant d'un processeur non inhibé se fait en dernier, donc a la moins forte priorité. Pour le reste, exceptée la mise en file d'attente différenciée selon le type de demande, l'algorithme de gestion du nouveau cache est analogue à celui qui a été exposé précédemment. Le lecteur trouvera en annexe III le modèle écrit en LASCAR de cette deuxième version du cache.

VII.2 RESULTATS DE SIMULATION

Mis à part le remplacement de l'ancien modèle de cache, tous les paramètres de la simulation sont restés identiques. Le tableau 3 résume les mesures recueillies à l'arrêt du premier processeur et à l'arrêt du dernier processeur. Ces résultats sont à comparer avec ceux de la dernière colonne des tableaux 1 et 2.

Examinons tout d'abord ce qui s'est passé lorsque le premier processeur arrive en fin de trace. On note que le processeur a, en moyenne, attendu 2 cycles d'horloge de moins en lecture, et 9 cycles d'horloge de moins en écriture. Par contre, le temps d'attente pour un test and set s'est considérablement allongé : passant de 320 cycles à 796, il a été multiplié par un facteur 2,5. Le bilan est cependant positif; la trace a été exécutée en 1000 cycles de moins, ce qui représente un gain de temps de 6 %. En ce qui concerne la molécule, sa puissance, calculée en nombre d'instructions par seconde, passe de 1317.10^3 à 1437.10^3, d'où un accroissement de performances de 8 %.

Il faut cependant noter que la complexité accrue du cache implique un coût supplémentaire en matériel, qui doit être évalué, et qui seul confirmera (ou infirmera) l'opportunité du choix de l'une ou l'autre version du cache.

TABLEAU 3

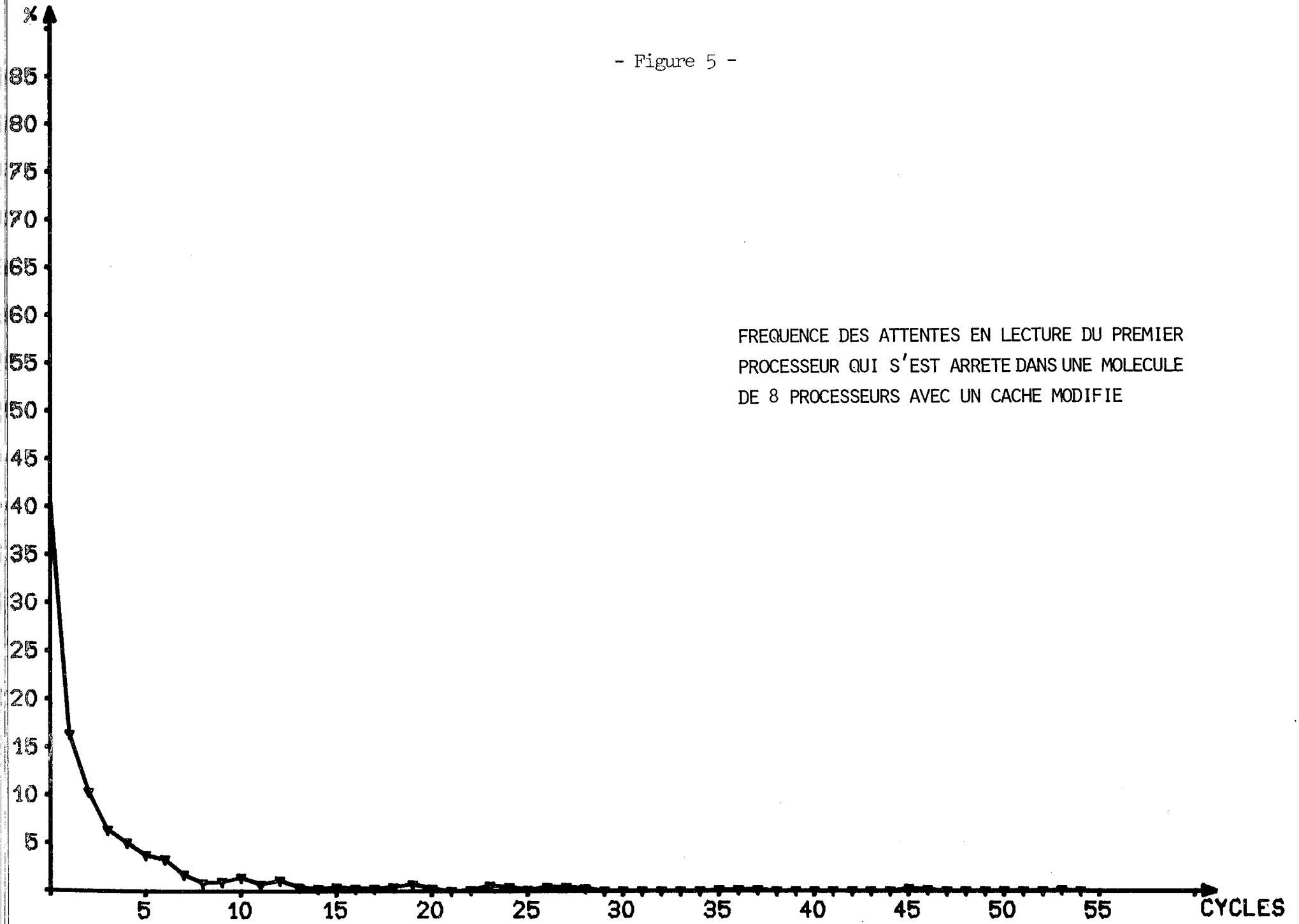
8 PROCESSEURS - CACHE MODIFIE	Arrêt premier processeur	Arrêt dernier processeur
Molécule		
. nombre de cycles simulés	16 826	20 716
. nombre d'instructions exécutées par la molécule	3 869	4 045
. puissance de la molécule en nombre d'instructions par seconde	1437.10^3	1220.10^3
. nombre moyen de cycles par instruction	4,29	5,12
Mémoire principale		
. nombre de lectures	515	540
. nombre d'écritures	828	860
. nombre de test and set	11	16
. % temps d'inactivité	35,2 %	44,9 %
Cache		
. longueur moyenne FDM	1,56	1,30
. longueur moyenne FDINV	0,79	1,33
. % temps d'inactivité	0,82 %	0,67 %
. inter-temps moyen appels mémoire	6,33	6,85
Processeur		
. nombre d'instructions exécutées	507	505
. puissance (nombre d'instructions par seconde)	188.10^3	152.10^3
. attente moyenne en lecture	3,41	3,38
. attente moyenne en écriture	19,9	19,43
. attente moyenne sur test and set	796	1337
. % temps d'attente	39 %	44 %
. nombre de cycles par instruction	33,18	41,02

Le gain de puissance de la molécule se répercute sur l'activité du cache et de la mémoire principale. Le cache est maintenant saturé, son taux d'activité dépasse 99 %. De manière encore plus évidente que précédemment, le nombre de processeurs qu'il est envisageable de connecter à un même cache ne peut dépasser huit. La mémoire principale voit son taux d'inactivité tomber de 51 % à 35 %, ce qui indique que la molécule, à sa puissance maximale (tous les processeurs occupés et aucun temps d'attente), est à elle seule responsable d'un fort débit sur le bus mémoire.

Si nous comparons les figures 2 et 5 qui donnent la fréquence des attentes en lecture, nous notons une allure tout à fait semblable des deux courbes, celle de la figure 5 se situant légèrement au-dessus de l'autre pour les points d'abscisse comprise entre 2 et 12 cycles. Une comparaison analogue des figures 3 et 6, qui donnent la fréquence des attentes en écriture, révèle des différences plus notables. Eliminons les 2 % d'attentes de 2 cycles qui correspondent à la ré-écriture du test-and-set, cette ré-écriture ayant été englobée dans les temps d'attente sur test and set dans la deuxième simulation. Les attentes minimales, de 8 à 10 cycles ne varient guère : 17 % et 18 %. Mais les attentes de 11 à 20 cycles passent de 30 % sur la figure 3 à 35 % sur la figure 6, celles de 21 à 30 cycles passent de 22 % sur la figure 3 à 29 % sur la figure 6. Il y a, pour la deuxième simulation, un tassement de la courbe des attentes en écriture vers les faibles abscisses.

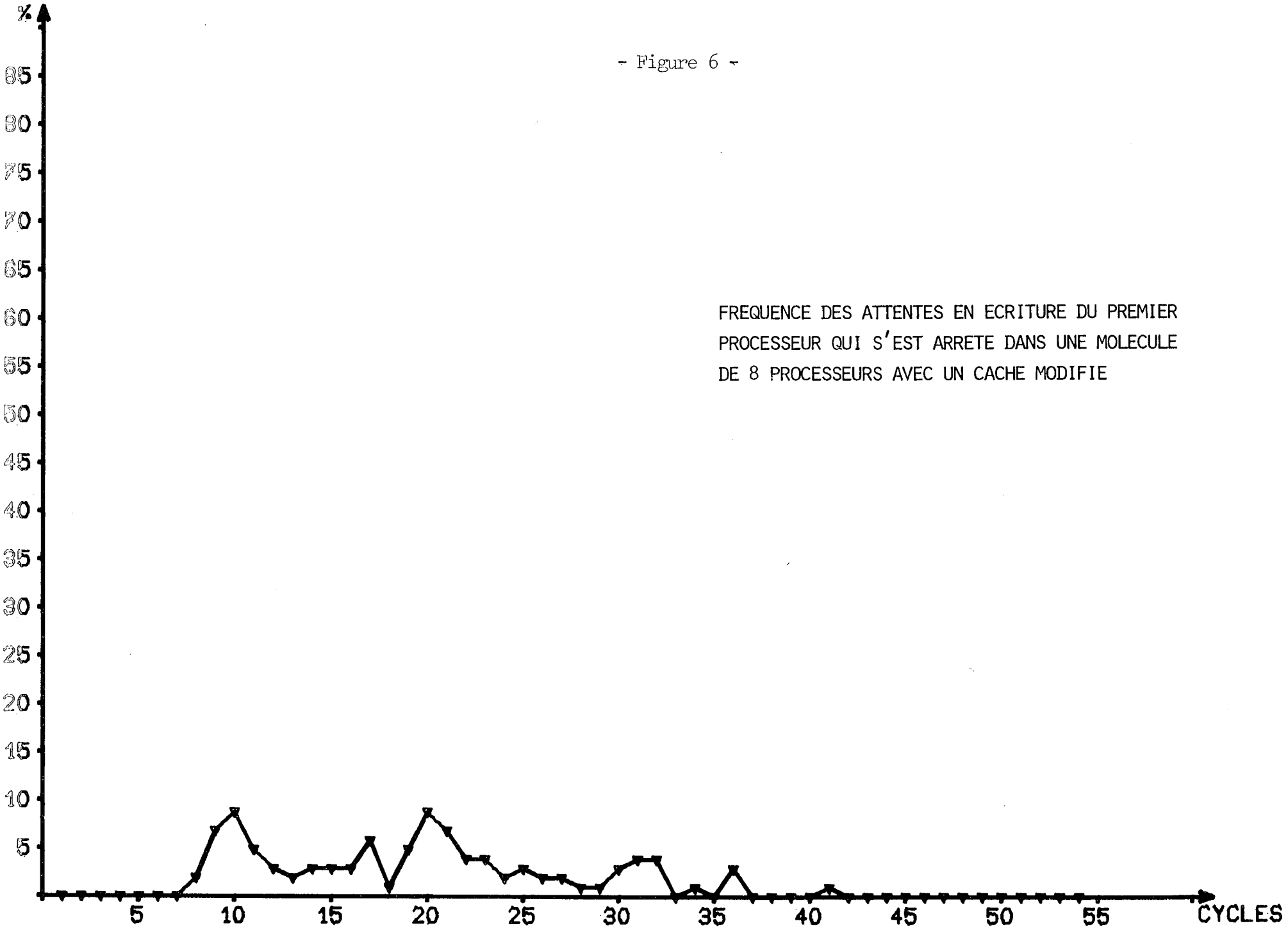
Le gain de performance a donc été obtenu essentiellement sur les demandes d'écriture, ce qui était prévisible.

- Figure 5 -



FREQUENCE DES ATTENTES EN LECTURE DU PREMIER PROCESSEUR QUI S'EST ARRETE DANS UNE MOLECULE DE 8 PROCESSEURS AVEC UN CACHE MODIFIE

- Figure 6 -



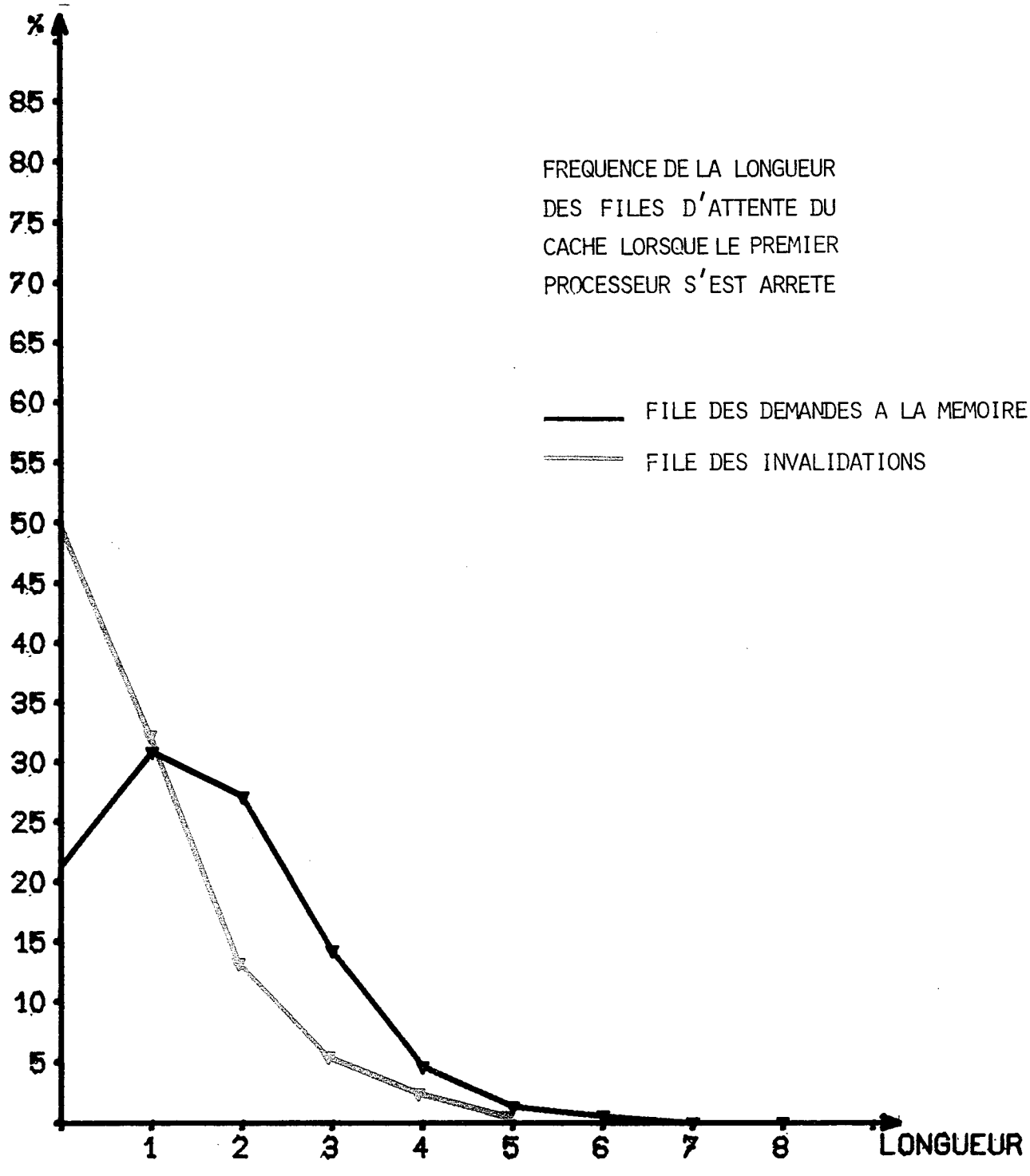
FREQUENCE DES ATTENTES EN ECRITURE DU PREMIER PROCESSEUR QUI S'EST ARRETE DANS UNE MOLECULE DE 8 PROCESSEURS AVEC UN CACHE MODIFIE

L'histogramme de la longueur des files d'attentes FDM et FDINV a été relevé. La courbe noire de la figure 7, représente l'histogramme de la longueur de FDM : cette longueur ne dépasse jamais 6, avec une valeur moyenne de 1,5. Cela signifie que, dans le pire des cas, un processeur qui émet une demande nécessitant un échange avec la mémoire principale attendra 60 cycles d'horloge ; ce qui confirme, dans les mesures de temps d'attente en lecture/écriture, le fait que les courbes obtenues ne comportent plus de point isolé de très forte abscisse.

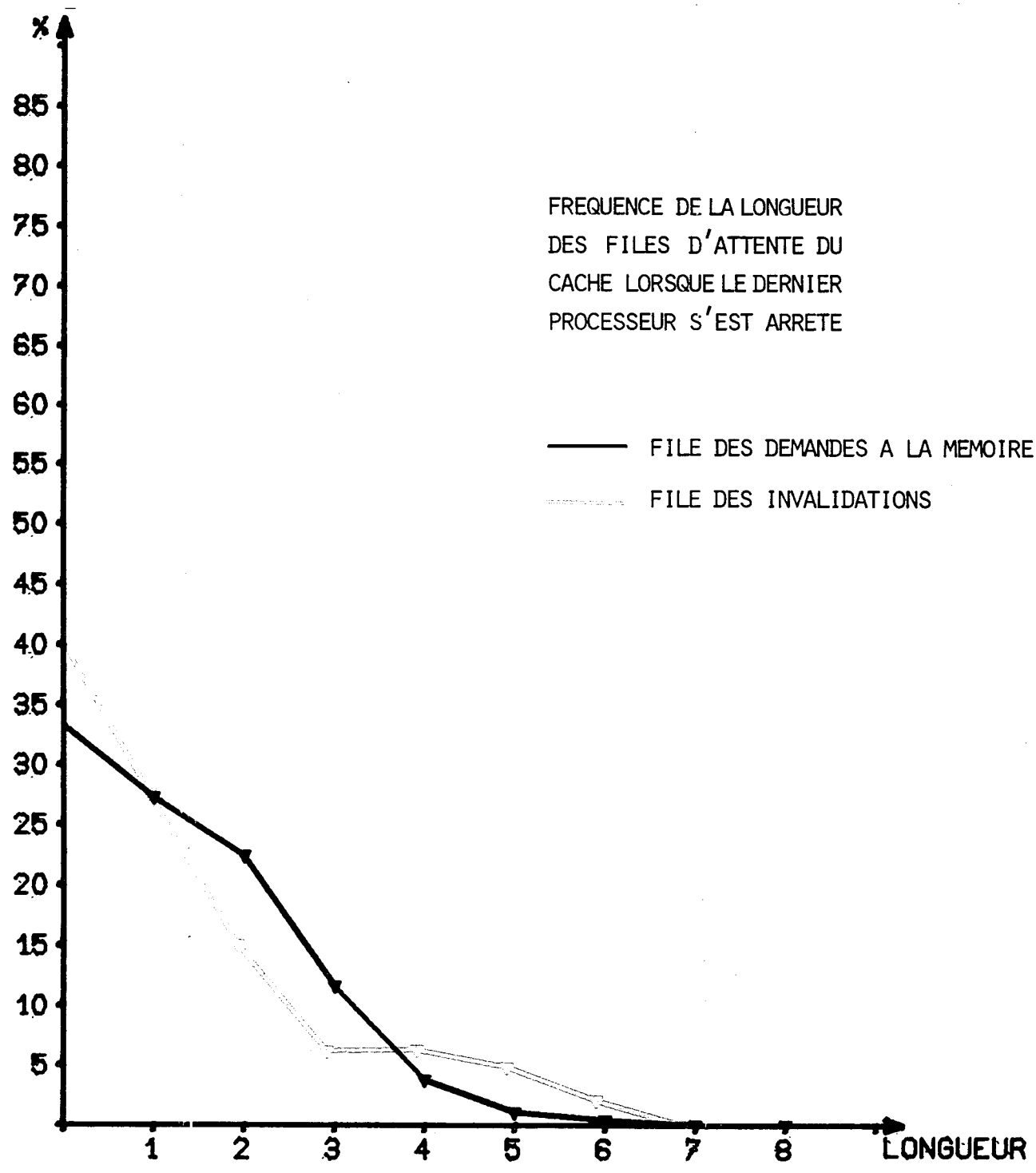
La file des demandes d'invalidation a une probabilité 1/2 d'être vide, ce qui signifie qu'une invalidation partielle du cache est en cours 50 % du temps. Ce phènomène est évidemment très lié au nombre important de lignes du cache.

L'analyse des résultats à l'arrêt du dernier processeur révèle les mêmes tendances : accroissement de puissance du processeur (+ 9 %) et de la molécule (+ 9 %), diminution des temps d'attente en lecture et en écriture, temps d'attente sur test and set multipliés par un facteur 3.

L'histogramme de la longueur de FDM et de FDINV (figure 8) est par contre sensiblement différent. En effet, vers la fin de la simulation, tous les processeurs arrivent en fin de trace et envoient une demande d'invalidation. La file FDINV se remplit alors, tandis que FDM se vide.


%
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
1
2
3
4
5
6
7
8
LONGUEUR
FREQUENCE DE LA LONGUEUR
DES FILES D'ATTENTE DU
CACHE LORSQUE LE PREMIER
PROCESSEUR S'EST ARRETE
FILE DES DEMANDES A LA MEMOIRE
FILE DES INVALIDATIONS

- Figure 7 -


%
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
1
2
3
4
5
6
7
8
LONGUEUR
FREQUENCE DE LA LONGUEUR DES FILES D'ATTENTE DU CACHE LORSQUE LE DERNIER PROCESSEUR S'EST ARRETE
FILE DES DEMANDES A LA MEMOIRE
FILE DES INVALIDATIONS

- Figure 8 -

CONCLUSION

Les simulations que nous venons de présenter sont des simulations extrêmement fines, puisqu'elles se situent au niveau du cycle d'horloge. Leur coût nous a contraint à en limiter le nombre. En effet, la mise au point du modèle, les essais préliminaires et les simulations proprement dites représentent 5 heures d'unité centrale sur l'ordinateur IBM 360/67.

L'étude qui a été faite est donc très partielle. Cependant, les résultats obtenus permettent de dégager un certain nombre de conclusions intéressantes et ouvrent la voie vers des études complémentaires.

D'autres objectifs, qui ne sont pas apparus dans ces simulations, notamment des objectifs de fiabilité et de facilité de reconfiguration du système, ont présidé à l'élaboration de l'architecture du multi-processeur. En particulier, il est prévu que chaque processeur déroule assez fréquemment des micro-programmes de test de bon fonctionnement, pendant lesquels il n'a aucune influence sur la molécule. Il faudra donc procéder à des simulations de molécules contenant 6 ou 7 processeurs, afin de tenir compte de la présence d'un processeur supplémentaire en train de procéder au test de ses propres circuits. Il est très vraisemblable que le coût d'un processeur sera bien inférieur à celui du cache. Le concepteur du matériel aura alors la charge, à partir de l'étude que nous avons menée, de déterminer le nombre de processeurs par molécule qui assure le meilleur compromis coût/performance.

Le nombre de molécules qu'il est possible de connecter au bus de la mémoire principale dépend du nombre de processeurs par molécule. Les simulations qui ont été faites concernent les échanges mémoire en régime permanent. Une étude des régimes transitoires, en particulier du chargement du cache après invalidation, sera par la suite nécessaire. Cependant, même en régime permanent, les taux d'occupation de la mémoire principale montrent que, si l'on veut que la structure permette plus de 2 molécules de 8 processeurs (ou plus de 3 molécules de 4 processeurs), il faut prévoir une mémoire principale découpée en bancs, avec accès indépendant aux différents bancs.

La simulation a mis en évidence le caractère extrêmement pénalisant du mécanisme d'invalidation, qui assure la cohérence des informations entre plusieurs caches et la mémoire principale, lorsque le cache contient un grand nombre de lignes. Lorsque les demandes sont mises dans une seule file d'attente, tous les processeurs sont ralentis par l'émission d'un test and set par l'un d'eux. L'introduction d'une deuxième file d'attente rajoute du matériel, et ne permet d'accroître la performance de la molécule que de 9 %, en ne ralentissant que le processeur responsable de l'invalidation du cache ; une évaluation du coût de cette complexité accrue du cache établira lequel des deux algorithmes est le plus rentable. Il est cependant plus probable qu'une révision de l'organisation du cache, en diminuant le nombre de lignes et soit multipliant le nombre de colonnes, soit augmentant la taille du bloc, sera de ce point de vue plus efficace. Cet aspect de l'étude du cache, mettant en jeu des traces d'instructions gigantesques, relève d'une modélisation plus abstraite.

Ou bien, il faudra imaginer et évaluer une autre solution au problème de la cohérence des informations partagées entre différents caches.
Cette étude est en cours.

TROISIEME PARTIE

PROGRAMMES DE TRAITEMENT
EVALUATION

I . LE "SYSTEME" LASCAR

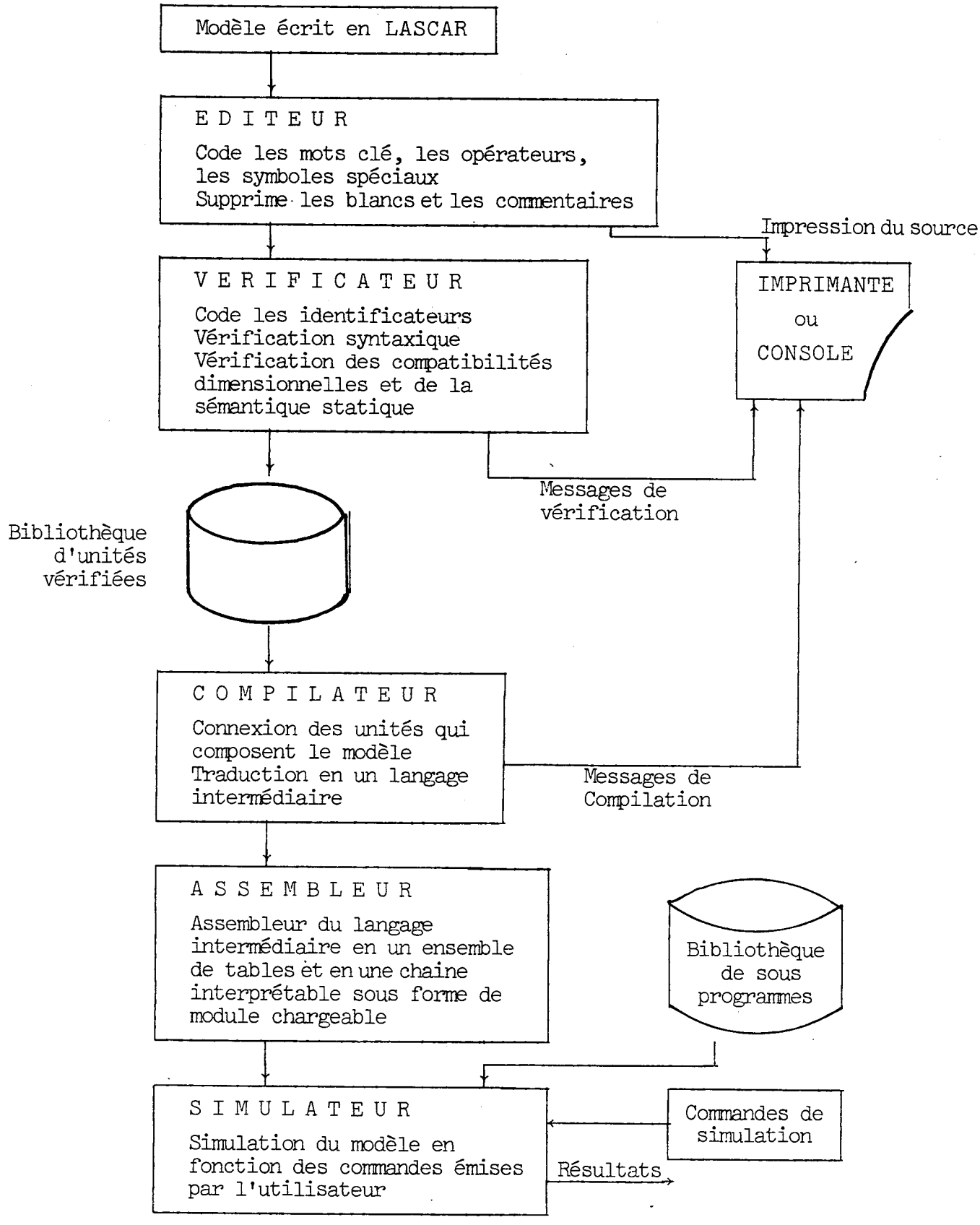
L'ensemble des programmes de traitement d'une description écrite en LASCAR est un sur-ensemble de la version synchrone du système CASSANDRE, et est organisé selon un schéma identique à celui-ci. Ainsi, toute description écrite en CASSANDRE est acceptée, sans qu'aucune modification soit nécessaire, par le système LASCAR ; et il est possible de faire coexister des unités écrites en CASSANDRE pur avec des unités écrites de manière fonctionnelle au sein d'un même modèle, ce qui était l'un des objectifs que nous nous étions fixés.

La traduction d'une description écrite en LASCAR est effectuée en quatre passages, par des modules qui se recouvrent en mémoire principale. Le résultat de cette traduction est ensuite chargé avec le simulateur ; l'utilisateur dispose à ce niveau d'un langage de commande lui permettant de diriger la simulation.

Nous avons donc repris, pour écrire le système LASCAR, la structure du système CASSANDRE, et réutilisé, en les étendant, les mécanismes de compilation et de simulation. Ainsi, à la notion de bibliothèque d'unités syntaxiquement correctes issues du vérificateur, nous avons adjoint la notion de bibliothèque de sous-programmes pré-compilés que l'on charge en mémoire avec le simulateur et le modèle traduit issu de l'assembleur. Par ailleurs, le langage de commande du simulateur a été étendu pour y inclure les ordres de rangement, d'impression et de test des variables entières.

Pour le reste, la réalisation des modules de traitement du langage LASCAR nous a conduit à apporter d'innombrables modifications ponctuelles aux modules du système CASSANDRE, à rajouter de nombreuses règles de grammaire dans l'éditeur, le vérificateur et le compilateur, et à étendre tous les modules par un ensemble de fonctions complémentaires ; cependant, la présentation externe du système est restée inchangée, ce qui est sans doute un avantage pour l'utilisateur. Et les pages III.2 à III.10 qui suivent peuvent être considérées comme des rappels de ce qui avait été publié dans [A2] et [A3].

Le système LASCAR, qui a été réalisé en collaboration avec M. GRABOWIECKI, est écrit en assembleur et macro-assembleur IBM 360. Il en existe une version conversationnelle, implantée sous les systèmes CP/CMS (IBM 360/67) et VM (IBM 370), et une version en traitement par lots sous le système OS (IBM 360-370).


Modèle écrit en LASCAR
E D I T E U R
Code les mots clé, les opérateurs, les symboles spéciaux
Supprime les blancs et les commentaires
Impression du source
V E R I F I C A T E U R
Code les identificateurs
Vérification syntaxique
Vérification des compatibilités dimensionnelles et de la sémantique statique
IMPRIMANTE ou CONSOLE
Messages de vérification
Bibliothèque d'unités vérifiées
C O M P I L A T E U R
Connexion des unités qui composent le modèle
Traduction en un langage intermédiaire
Messages de Compilation
A S S E M B L E U R
Assembleur du langage intermédiaire en un ensemble de tables et en une chaine interprétable sous forme de module chargeable
Bibliothèque de sous programmes
S I M U L A T E U R
Simulation du modèle en fonction des commandes émises par l'utilisateur
Commandes de simulation
Résultats

La succession des quatre programmes de traduction d'une description écrite en LASCAR en un fichier binaire chargeable prend de quelques secondes, pour les modèles simples, à quelques dizaines de secondes, pour les modèles très longs et contenant de nombreuses imbrications d'unités. Par contre, le temps de simulation se compte en minutes, et même en heures, comme nous l'avons vu dans la deuxième partie de ce mémoire, lorsque l'on veut étudier le fonctionnement d'une machine complexe pendant un nombre important d'unités de temps. Il nous a donc semblé tout à fait nécessaire de prélever des mesures sur le simulateur, afin d'avoir une image précise de son comportement, et de pouvoir éventuellement en tirer des conclusions pour l'élaboration d'un simulateur optimisé.

II . LE SIMULATEUR

L'utilisateur qui veut simuler sa description charge en mémoire le simulateur, le fichier binaire issu de l'assembleur et les sous-programmes appelés dans la description. Il dispose d'un langage de commande qui lui permet de :

- se positionner dans l'environnement d'une unité de la description

  Exemple : UNITE PCACHE

- charger les éléments de mémorisation de la description, et les signaux et horloges d'entrée de l'unité englobante

  Exemple :

```
UNITE  MODELE
RANGE  H = 1              chargement d'une horloge
UNITE  PCACHE
RANGE  INHIBE (1:4) = 1111 chargement d'un registre
IND    LMEM = 255         chargement d'un entier en décimal
INH    INV  =  FF         chargement d'un entier en hexadécimal
```

- faire imprimer la valeur de n'importe quelle variable du modèle

  Exemple :

```
OUTD  PTFDM          impression d'un entier en décimal
OUTB  FDM(1)         impression d'un entier en binaire
```

- tester la valeur de n'importe quelle variable du modèle

  Exemple :

  ```
  SI INV     =  10  ALORS      test d'un entier
  SI PCACHE  =  LIRE  ALORS    test du registre d'état
  ```

- faire exécuter un ou plusieurs cycles de calcul

  Exemple :

  ```
  CYCLE  50      exécution de 50 cycles
  ```

- sauvegarder l'état du modèle en vue d'une poursuite ultérieure de la simulation, ou récupérer une sauvegarde (version conversationnelle seulement).

  Exemple :

  ```
  SAUVE    12
  RESTORE   1
  ```

Ces ordres peuvent être émis individuellement, ou être regroupés dans des fichiers pour constituer des procédures de commande. Dans ce deuxième cas, il peut être utile d'introduire des instructions d'itération et de rupture de séquence, qui font aussi partie du langage de commande.

Dans la version conversationnelle, l'utilisateur envoie ses requêtes depuis sa console, et reçoit les réponses sur sa console également. Dans la version en traitement par lots, toutes les commandes de simulation sont regroupées dans un fichier d'entrée du simulateur (cartes ou disque) et les impressions sont dirigées vers l'imprimante. Le lecteur trouvera en annexe un exemple de simulation conversationnelle de l'unité PCACHE.

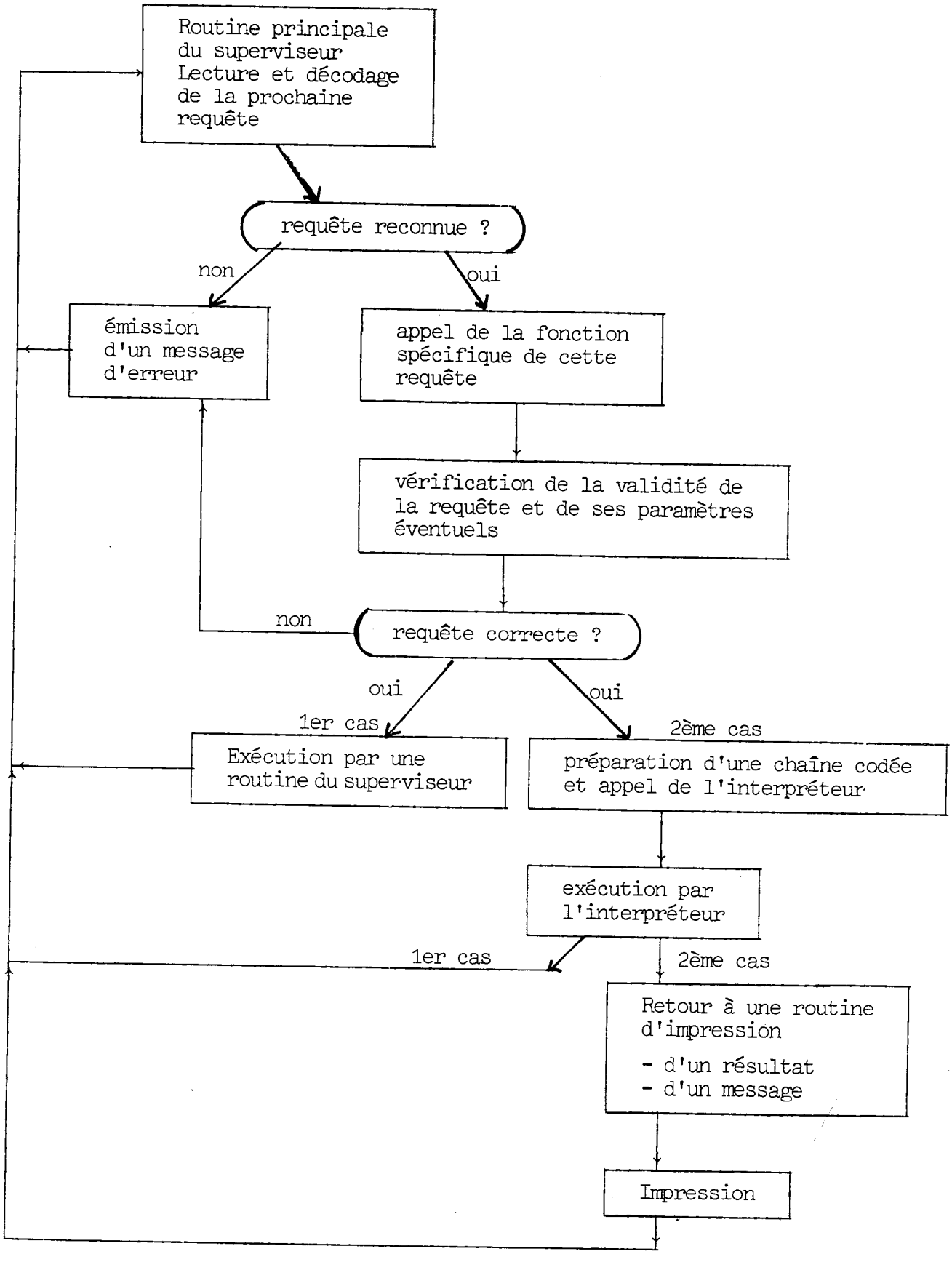
Le simulateur est divisé en deux modules de taille équivalente : un superviseur et un interpréteur.

II.1 LE SUPERVISEUR DE SIMULATION

Le superviseur de simulation assure l'interface entre la description et le monde extérieur. Au démarrage de la simulation, il procède à toutes les initialisations, puis se met en attente de commande de l'utilisateur (version conversationnelle) ou va chercher la première commande de son fichier d'entrée (version en traitement par lots).

Toute requête émise par l'utilisateur est décodée par la routine principale du superviseur. Si cette requête est reconnue, une routine spécifique vérifie sa validité et la validité des paramètres éventuels ; et suivant les cas l'exécute directement, ou construit une "pseudo chaîne" et fait appel à l'interpréteur de simulation. Le contrôle est rendu soit à la routine principale qui décodera la prochaine requête, soit préalablement à une routine d'impression de résultat ou de message d'erreur.

L'organigramme général du superviseur de simulation est le suivant :


Routine principale du superviseur Lecture et décodage de la prochaine requête
requête reconnue ?
non
oui
émission d'un message d'erreur
appel de la fonction spécifique de cette requête
vérification de la validité de la requête et de ses paramètres éventuels
non
requête correcte ?
oui
oui
1er cas
2ème cas
Exécution par une routine du superviseur
préparation d'une chaîne codée et appel de l'interpréteur
exécution par l'interpréteur
1er cas
2ème cas
Retour à une routine d'impression
- d'un résultat
- d'un message
Impression

II.2 STRUCTURE INTERNE D'UNE DESCRIPTION

La représentation interne d'une description écrite en LASCAR est composée d'un ensemble de tables et d'une chaîne interprétable.

- TABLEM1 et TABLEM2

Ces deux tables sont identiques. Elles contiennent les variables de type registre et état. Au cours d'un cycle de calcul, lors de l'évaluation d'une expression en partie droite d'une affectation de registre (ou de variable d'état) les valeurs prises en compte pour les registres (ou les variables d'état) sont leurs valeurs avant le top d'horloge, rangées dans TABLEM1. Les nouvelles valeurs sont stockées dans TABLEM2. A la fin du cycle, TABLEM2 est reversée dans TABLEM1.
Ce mécanisme assure donc que les affectations de registres et de variables d'état sont effectuées de manière collatérale.

- TABLEI

Elle contient les variables de type signal et horloge, qui sont des variables immédiates, et les variables de type compteur, entier et tableau entier sur lesquelles les calculs sont séquentiels.

- SYMB

C'est la table de correspondance entre les noms symboliques des variables, tels qu'ils sont déclarés dans la description, et leur adresse d'implantation dans TABLEM1 ou dans TABLEI. Elle est indispensable à la prise en compte des ordres de simulation pour lesquels les noms des êtres référencés sont leurs noms symboliques (ordres de chargement de valeur et d'impression en particulier).

- DUPL

Elle contient pour chaque unité de la description le nom et les numéros de duplication des unités englobées, et permet à l'utilisateur de se placer en cours de simulation dans l'environnement d'une unité particulière.

- TABLE

C'est le graphe d'appel des unités, pour une description écrite sous la forme d'une arborescence d'unités emboitées. Les chaînages sont effectués dans l'ordre inverse de ceux qui sont indiqués par DUPL.

- CHAINE

C'est le reflet, sous la forme d'une suite d'ordres interprétables en notation post-fixée, des opérations nécessaires à la simulation des instructions d'un modèle écrit en LASCAR.

II.3 L'INTERPRETEUR DE SIMULATION

L'interpréteur de simulation a pour mission d'interpréter les ordres constituant la chaîne de simulation de la description, et les ordres générés dans une "pseudo chaîne" par le superviseur pour l'exécution de certaines requêtes de l'utilisateur. Il est composé d'un décodeur et d'un ensemble de 135 routines qui travaillent à l'aide de 2 piles (une pile de mots pour le calcul des expressions entières, une pile de demi-mots pour tous les autres calculs). Le décodeur, en fonction du code du prochain ordre à traiter dans la chaîne, sert d'aiguillage et donne le contrôle à la routine de traitement appropriée. Un code particulier, indiquant une fin de séquence, permet le retour au superviseur de simulation.

La routine principale de l'interpréteur est comme dans l'interpréteur CASSANDRE la routine CYCLE. Elle est appelée lorsque l'utilisateur demande la simulation d'un ou plusieurs cycles d'horloge. Reflet de l'algorithme de simulation, la routine CYCLE pilote le parcours de la chaîne interprétable. Nous en rappelons le principe :

1 - Tours en S

On exécute les instructions de connexion de signaux et d'horloges, autant de fois que nécessaire pour stabiliser leurs valeurs. Si le nombre de tours dépasse une valeur maximale fixée, la description est déclarée instable et un message d'erreur est envoyé.

2 - Tour en R

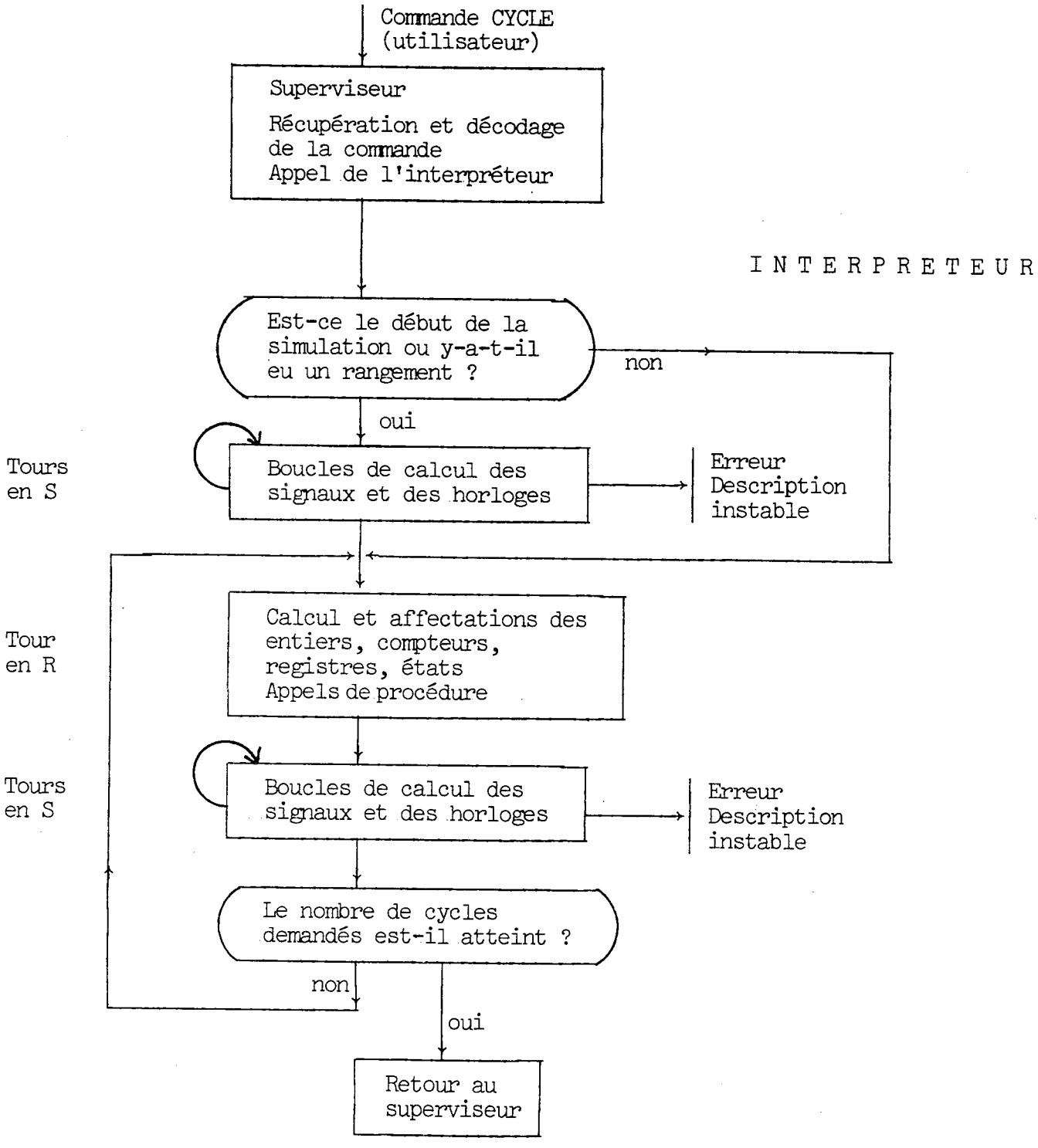
On exécute une fois les instructions d'affectation de registres, d'états, d'entiers, de compteurs, et les appels de procédure.

3 - Tours en S

Les valeurs des signaux en sortie de registres ou d'entiers doivent être ré-ajustées en fonction des affectations qui viennent d'avoir lieu.
Il faut donc procéder à de nouveaux balayages de la chaîne de simulation, pour recalculer les signaux et les stabiliser.

L'algorithme est effectué dans sa totalité lors de l'exécution du premier cycle de la simulation, et lorsque l'utilisateur a modifié, par une commande de rangement, la valeur d'au moins une variable de la description. Par contre, si plusieurs cycles sont demandés en une même commande, ou si aucun rangement n'a été fait entre deux demandes de cycle, la première série de tours en S est inutile, et par conséquent n'est pas exécutée.

L'organigramme qui suit décrit l'algorithme de simulation d'un (ou plusieurs) cycle de calcul.


Commande CYCLE
(utilisateur)
Superviseur
Récupération et décodage
de la commande
Appel de l'interpréteur
INTERPRETEUR
Est-ce le début de la
simulation ou y-a-t-il
eu un rangement ?
non
oui
Tours
en S
Boucles de calcul des
signaux et des horloges
Erreur
Description
instable
Tour
en R
Calcul et affectations des
entiers, compteurs,
registres, états
Appels de procédure
Tours
en S
Boucles de calcul des
signaux et des horloges
Erreur
Description
instable
Le nombre de cycles
demandés est-il atteint ?
non
oui
Retour au
superviseur

II.4 MINIMISATION DES "TOURS EN S"

Une première remarque vient immediatement à l'esprit, à la lecture de cet algorithme : le temps de simulation est directement fonction du nombre de tours de stabilisation des signaux et des horloges (tours en S). Monsieur Hugues JACOLIN [F13] a établi sous quelles conditions il est possible de ré-ordonner statiquement les instructions de connection d'une description pour minimiser le nombre de tours de stabilisation. Nous illustrerons sur un exemple extrèmement simple le principe sur lequel est basée son étude.

Soit la portion de description :

```
registre D;
signal A,B,C;
A := B;  B := C + D ;
```

Initialement, toutes les variables sont nulles. Supposons que nous rangeons la valeur 1 dans D. Au premier tour de stabilisation, A reste à 0 et B prend la valeur 1. Au deuxième tour, A prend la valeur 1, B reste inchangé. Un troisième tour permet de vérifier que toutes les variables ont atteint leur valeur stable.

Si par contre nous avions écrit les instructions de connexion dans l'ordre inverse, c'est à dire :

```
registre D ;
signal A, B, C ;
B := C + D ;  A := B ;
```

seuls un tour de stabilisation et un tour de vérification auraient été nécessaires. De plus, dans ce cas, il est démontré que toutes les variables reçoivent leur valeur définitive en un seul parcours de la chaine à interpréter, ce qui permet de s'abstenir d'effectuer le tour de vérification.

Il existe cependant des cas pour lesquels il n'est pas possible d'ordonner les instructions de manière à obtenir une stabilisation en un seul passage. Citons notamment :

- les instructions dans lesquelles une même variable se trouve en partie gauche et en partie droite du symbole de connexion.

Exemple

Description d'un compteur 8 bits en CASSANDRE

```
signal A (0:7) ;
registre R (0:7) ;
horloge H ;
A := * D |1| ( A . R ) Ɛ 1 ;
< H > R <=  R ≠ A ;
```

- Les instructions faire qui ont comme paramètre une variable d'état dont la valeur n'est connue qu'à la simulation.

Exemple

Tiré de l'unite PCACHE, l'état SELECT :

```
00709   SELECT: 'SIE' AIG#2 'ALORS' OK:=000000000000;
00739       <H> DEMAND $= $(INHIBE(1:12).(/+DEM(0:4,))),
00779           'SIE' DEMAND = 0 'ALORS'
00789   ('SI' ACCESM 'ALORS'
00798       (AIG$='SIE' REPMEM=0 'ALORS' 1C.ACC1
00820                'SINON' REPMEM,
00828           TRAV$='SIE' AIG=10 'ALORS' FDM1(1)
00848                'SINON' FDM(1))
00856    'SINON'
00857      ('SIE' PTFDM  > 0 'ALORS' (AIG$=4,TRAV$=FDM(1))
00886     'SINON'
00887   ('SIE' ACC1=0 'ALORS'
00896       (AIG$=0,'P 1'(INACTIF),INHIBE(1:8)<=11111111)
00935         'SINON' (TRAV$=FDM1(1),AIG$=10))))
00960           'SINON'
00961                    ('CALL' SELECTE (CIRCUL,NBP,DEMAND),
00990                     'SI' DEM(,CIRCUL) 'ALORS'
01004                          (AIG $= 5)
01011                          (AIG $= 6)
01018                          (AIG $= 7)
01025                          (AIG $= 8)
01032                          (AIG $= 9 ));
01041           'FAIRE' ECLATM(AIG);
```

En définitive, si certaines descriptions se prêtent bien à une ré-organisation des instructions de connexion assurant un nombre minimal de tours de stabilisation (un programme spécifique, que l'on exécute entre le vérificateur et le compilateur, effectue ce traitement), il n'est pas possible de garantir ce type d'optimisation dans le cas général, à moins d'imposer des restrictions importantes sur le langage.

Il apparait nécessaire de revoir le problème des performances du simulateur de manière plus globale. Dans cette optique, il est indispensable de prélever des mesures sur son fonctionnement.

III. LES MESURES

III.1 QUE MESURER ?

L'emploi de la version conversationnelle du système LASCAR doit correspondre à la phase d'écriture et de mise au point d'un modèle. Notre expérience, et celle d'autres utilisateurs, nous a prouvé que, dans ces conditions, les ordres du langage de commande les plus utilisés sont le positionnement dans l'environnement d'une unité de la description, et surtout les ordres de rangement et d'impression de valeurs. En revanche, le nombre de cycles simulés est assez faible, au plus quelques centaines, plus généralement quelques dizaines; et le nombre de cycles demandés en une seule commande est petit, de l'ordre de quelques unités.
Ce type d'utilisation du simulateur est donc gros consommateur d'entrées-sorties, ce qu'attestent les différences notables entre le temps virtuel (temps de calcul, ou temps CMS) et le temps réel (temps de calcul + traduction des programmes canaux, ou temps CP) consommés. Ce n'est donc pas pour ces simulations en pas à pas, dont le coût est fortement grevé par les appels au système superviseur, chaque commande impliquant au moins deux entrées-sorties à la console, qu'il faut tenter une optimisation des temps de simulation.
Par contre, les simulations longues, qui utilisent la version en traitement par lots, sont extrêmement coûteuses en temps de calcul de l'unité centrale. Pour ces simulations, les ordres de rangement servent essentiellement à l'initialisation du modèle, et les ordres d'impression de résultats partiels ne sont émis qu'après la demande d'un nombre élevé de cycles. Aussi les commandes de cycles sont-elles des commandes portant sur plusieurs milliers, voire plusieurs dizaines de milliers de cycles. C'est donc lors de la simulation ininterrompue de plusieurs milliers de cycles d'horloge qu'il est important de connaitre le fonctionnement du simulateur, et que l'on peut espérer formuler des modifications conduisant à une amélioration significative des temps de simulation. En conséquence, c'est essentiellement l'interpréteur de simulation que nous avons mesuré.

III.2 EVALUATION DES ROUTINES DE L'INTERPRETEUR

Les routines de l'interpréteur de simulation sont de complexité très disparate: de quelques lignes à plusieurs pages d'assembleur. Mais la difficulté essentielle provient du fait que certaines ont un temps d'exécution fixe, d'autres non:

- le décodeur, les routines de rupture de séquence dans la chaine, les routines traitant les variables scalaires, l'interprétation des opérateurs (arithmétiques et de comparaison) appliqués à des variables scalaires déroulent toujours le même nombre d'instructions; il suffit de les compter.
- les routines d'empilement, de rangement, d'interprétation des opérateurs (logiques, arithmétiques, de comparaison, de conversion, de transformation qui ont comme paramètre des variables de dimensions arbitraires déroulent un nombre d'instructions qui dépend des dimensions de leur paramètre .

Pour évaluer les routines de la deuxième catégorie, deux méthodes ont été employées.

- Lorsque cela était possible, nous avons établi, par comptage, en tenant compte des paramètres qui influent sur les répétitions de séquences d'instructions, une formule exacte permettant de calculer, pour chaque appel de la routine, le nombre d'instructions executées.
- Pour quelques routines extrèmement touffues, notamment celles, héritées de CASSANDRE, qui empilent ou rangent des sous-tableaux de variables booléennes, nous avons tracé, à l'aide de PILOTE, leur exécution lors de la simulation du multi-processeur, et avons établi un nombre typique d'instructions exécutées.

Le lecteur trouvera en ANNEXE IV la liste des routines de l'interpréteur de simulation, avec pour chacune :

son numéro
son nom
la fonction qu'elle réalise
le nombre exact d'instructions déroulées, ou la formule permettant de le calculer, lorsque cette information a pu être obtenue
le nombre typique d'instructions déroulées lors de la simulation du multi-processeur.

Cas particulier

Lors de l'appel d'une procédure Fortran ou assembleur, seules les instructions de la routine assurant l'interface et le passage de paramètres sont décomptées. Il est bien évident que les instructions de la procédure elle-même, étant indépendantes du compilateur LASCAR, ne nous intéressent pas ici. Par contre, il faudrait les prendre en considération dans une évaluation non plus du simulateur mais du modèle.

III.3 FREQUENCE D'APPEL DES ROUTINES DE L'INTERPRETEUR

Pour l'évaluation du comportement du simulateur, il est important de savoir quelles routines sont le plus fréquemment appelées, de manière à connaître le poids de chaque routine dans le coût total de la simulation. La fréquence d'appel des routines est évidemment très dépendante des instructions employées dans le modèle, et pourrait, à cet égard, servir à caractériser le "style" de programmation de l'utilisateur. A nouveau, nous avons utilisé la description du multi-processeur comme jeu d'essai, car c'était le modèle le plus complexe, et qui utilisait le plus grand nombre de primitives du langage, dont nous disposions.

Nous avons choisi de mesurer la version du modèle contenant deux processeurs. En effet, nous voulions faire apparaître le problème de la duplication d'une unité, mais nous ne voulions pas trop privilégier les instructions du processeur par rapport à celles des autres unités, notamment du cache qui utilise des constructions plus riches.

Un autre point doit être souligné : le modèle du multi-processeur a été écrit en toute connaissance des programmes de traitement du système LASCAR. Ainsi, dans chaque unité, l'ordre d'écriture des instructions tient compte du problème de la minimisation des tours en S à la simulation. De plus, l'utilisation de variables de type signal ou registre, exceptées les entrées-sorties des unités, a été rendue aussi réduite que possible : on ne trouve que 3 registres dans toute la description, et aucun signal autre que ceux servant à connecter les unités entre elles ; avant même de prélever des mesures, nous savions que l'emploi de signaux et de registres non scalaires était très coûteux.

Le lecteur ne devra donc pas être surpris par le fait que certaines routines n'ont jamais servi : tous les opérateurs n'apparaîssant pas dans la description. Les mesures que nous présentons n'ont donc pas la prétention d'être exhaustives. Tout au plus permettent-elles de dégager une tendance entre fréquence d'appel des routines longues et courtes, dans un programme faisant largement usage des primitives LASCAR.

Monsieur GRABOWIECKI a implanté, dans l'interpréteur de simulation, un mécanisme permettant de compter les appels de routines, et a modifié le superviseur de simulation de manière à ce que, lors de la reconnaissance de la commande FIN (commande d'arrêt de simulation), toutes les mesures recueillies soient rassemblées et imprimées dans un fichier avant l'arrêt de l'exécution.

Ce fichier est ensuite traité par un petit programme Fortran qui calcule, pour chaque routine, le pourcentage du nombre d'appels par rapport au nombre total d'appels de routines, et son coût pour la simulation (pourcentage du nombre d'instructions exécutées dans la routine par rapport au nombre total d'instructions exécutées dans l'interpréteur de simulation).

Le tableau ci-dessous donne les résultats obtenus pour la simulation, pendant 1000 cycles ininterrompus, du modèle de la molécule contenant deux processeurs. Cette simulation a coûté 70 secondes de calcul d'unité centrale.

NOMBRE TOTAL D'APPELS DE ROUTINES 1824512

NOMBRE TOTAL D'INSTRUCTIONS EXECUTEES DANS L'INTERPRETEUR, 35248397

Numéro	Nombre d'appels	Nombre d'instr. pour 1 appel	% du Nbre d'appels de routines	Nombre d'instr. exécutées dans la routine	% du Nbre total d'instr. exécutées
0	0	1	0.0	0	0.0
1	49223	5	2.698	246115	0.698
2	62869	5	3.446	314345	0.892
3	40072	7	2.196	280504	0.796
4	4120	4	0.226	16480	0.047
5	25352	4	1.390	101408	0.288
6	15208	16	0.834	243328	0.690
7	17368	16	0.952	277888	0.788
8	3802	12	0.208	45624	0.129
9	7868	35	0.431	275380	0.781
10	3934	5	0.216	19670	0.056
11	56708	5	3.108	283540	0.804
12	0	5	0.0	0	0.0
13	0	7	0.0	0	0.0
14	0	7	0.0	0	0.0
15	0	4	0.0	0	0.0
16	0	4	0.0	0	0.0
17	3934	6	0.216	23604	0.067
18	59044	3	3.236	177132	0.503
19	11403	4	0.625	45612	0.129
20	5901	2	0.323	11802	0.033
21	0	0	0.0	0	0.0
22	17535	30	0.961	526050	1.492
23	14735	3	0.808	44205	0.125
24	0	0	0.0	0	0.0
25	164552	4	9.019	658208	1.867
26	31472	7	1.725	220304	0.625
27	9185	8	0.503	73480	0.208
28	2553	35	0.140	89355	0.254
29	55397	46	3.036	2548262	7.229
30	4055	285	0.222	1155675	3.279
31	1724	47	0.094	81028	0.230
32	232	254	0.013	58928	0.167
33	5901	7	0.323	41307	0.117
34	1839	17	0.101	31263	0.089
35	33	45	0.002	1485	0.004
36	7862	74	0.431	581788	1.651
37	24013	337	1.316	8092381	22.958
38	74	73	0.004	5402	0.015
39	595	418	0.033	248710	0.706
40	1968	74	0.108	145632	0.413
41	0	0	0.0	0	0.0
42	0	8	0.0	0	0.0
43	0	15	0.0	0	0.0
44	29505	42	1.617	1239210	3.516
45	17703	75	0.970	1327725	3.767

Numéro	Nombre d'appels	Nombre d'instr. pour 1 appel	% du Nbre d'appels de routines	Nombre d'instr. exécutées dans la routine	% du Nbre total d'instru. exécutées
* 46 *	0 *	0 *	0.0 *	0 *	0.0
* 47 *	0 *	0 *	0.0 *	0 *	0.0
* 48 *	0 *	0 *	0.0 *	0 *	0.0
* 49 *	15736 *	74 *	0.862 *	1164464 *	3.304
* 50 *	0 *	0 *	0.0 *	0 *	0.0
* 51 *	0 *	0 *	0.0 *	0 *	0.0
* 52 *	1648 *	11 *	0.090 *	18128 *	0.051
* 53 *	4 *	12 *	0.000 *	48 *	0.000
* 54 *	0 *	0 *	0.0 *	0 *	0.0
* 55 *	0 *	0 *	0.0 *	0 *	0.0
* 56 *	1862 *	628 *	0.102 *	1169336 *	3.317
* 57 *	1967 *	136 *	0.108 *	267512 *	0.759
* 58 *	0 *	0 *	0.0 *	0 *	0.0
* 59 *	0 *	0 *	0.0 *	0 *	0.0
* 60 *	0 *	0 *	0.0 *	0 *	0.0
* 61 *	0 *	0 *	0.0 *	0 *	0.0
* 62 *	0 *	0 *	0.0 *	0 *	0.0
* 63 *	0 *	0 *	0.0 *	0 *	0.0
* 64 *	0 *	0 *	0.0 *	0 *	0.0
* 65 *	0 *	0 *	0.0 *	0 *	0.0
* 66 *	0 *	0 *	0.0 *	0 *	0.0
* 67 *	0 *	0 *	0.0 *	0 *	0.0
* 68 *	1000 *	27 *	0.055 *	27000 *	0.077
* 69 *	0 *	0 *	0.0 *	0 *	0.0
* 70 *	0 *	0 *	0.0 *	0 *	0.0
* 71 *	862 *	35 *	0.047 *	30170 *	0.086
* 72 *	0 *	14 *	0.0 *	0 *	0.0
* 73 *	0 *	14 *	0.0 *	0 *	0.0
* 74 *	5901 *	14 *	0.323 *	82614 *	0.234
* 75 *	0 *	14 *	0.0 *	0 *	0.0
* 76 *	0 *	14 *	0.0 *	0 *	0.0
* 77 *	0 *	14 *	0.0 *	0 *	0.0
* 78 *	0 *	8 *	0.0 *	0 *	0.0
* 79 *	0 *	8 *	0.0 *	0 *	0.0
* 80 *	3 *	35 *	0.000 *	105 *	0.000
* 81 *	4 *	21 *	0.000 *	84 *	0.000
* 82 *	57 *	3 *	0.003 *	171 *	0.000
* 83 *	5005 *	33 *	0.274 *	165165 *	0.469
* 84 *	1013 *	224 *	0.056 *	226912 *	0.644
* 85 *	4035 *	96 *	0.221 *	387360 *	1.099
* 86 *	7152 *	185 *	0.392 *	1323120 *	3.754
* 87 *	0 *	0 *	0.0 *	0 *	0.0
* 88 *	4207 *	6 *	0.231 *	25242 *	0.072
* 89 *	5 *	12 *	0.000 *	60 *	0.000
* 90 *	5901 *	17 *	0.323 *	100317 *	0.285
* 91 *	5901 *	581 *	0.323 *	3428481 *	9.727
* 92 *	3934 *	5 *	0.216 *	19670 *	0.056
* 93 *	0 *	4 *	0.0 *	0 *	0.0
* 94 *	0 *	13 *	0.0 *	0 *	0.0
* 95 *	4879 *	14 *	0.267 *	68306 *	0.194
* 96 *	12100 *	12 *	0.663 *	145200 *	0.412
* 97 *	10712 *	9 *	0.587 *	96408 *	0.274
* 98 *	2326 *	10 *	0.127 *	23260 *	0.066
* 99 *	0 *	0 *	0.0 *	0 *	0.0

Numéro	Nombre d'appels	Nombre d'instr. pour 1 appel	% du Nbre d'appels de routines	Nombre d'instr. exécutées dans la routine	% du Nbre total d'instr. exécutées
*100 *	186 *	12 *	0.010 *	2232 *	0.006
*101 *	0 *	0 *	0.0 *	0 *	0.0
*102 *	65 *	15 *	0.004 *	975 *	0.003
*103 *	0 *	0 *	0.0 *	0 *	0.0
*104 *	0 *	18 *	0.0 *	0 *	0.0
*105 *	0 *	18 *	0.0 *	0 *	0.0
*106 *	0 *	18 *	0.0 *	0 *	0.0
*107 *	0 *	18 *	0.0 *	0 *	0.0
*108 *	0 *	18 *	0.0 *	0 *	0.0
*109 *	432 *	18 *	0.024 *	7776 *	0.022
*110 *	6695 *	12 *	0.367 *	80340 *	0.228
*111 *	1154 *	12 *	0.063 *	13848 *	0.039
*112 *	37 *	21 *	0.002 *	777 *	0.002
*113 *	2830 *	21 *	0.155 *	59430 *	0.169
*114 *	12 *	21 *	0.001 *	252 *	0.001
*115 *	0 *	21 *	0.0 *	0 *	0.0
*116 *	1238 *	21 *	0.068 *	25998 *	0.074
*117 *	4169 *	21 *	0.228 *	87549 *	0.248
*118 *	8036 *	8 *	0.440 *	64288 *	0.182
*119 *	8110 *	10 *	0.445 *	81100 *	0.230
*120 *	1862 *	212 *	0.102 *	394744 *	1.120
*121 *	0 *	0 *	0.0 *	0 *	0.0
*122 *	0 *	0 *	0.0 *	0 *	0.0
*123 *	0 *	0 *	0.0 *	0 *	0.0
*124 *	0 *	0 *	0.0 *	0 *	0.0
*125 *	0 *	0 *	0.0 *	0 *	0.0
*126 *	2618 *	84 *	0.143 *	219912 *	0.624
*127 *	2400 *	82 *	0.132 *	196800 *	0.558
*128 *	74 *	26 *	0.004 *	1924 *	0.005
*129 *	0 *	9 *	0.0 *	0 *	0.0
*130 *	5041 *	10 *	0.276 *	50410 *	0.143
*131 *	7862 *	7 *	0.431 *	55034 *	0.156
*132 *	4617 *	13 *	0.253 *	60021 *	0.170
*133 *	262 *	87 *	0.014 *	22794 *	0.065
*134 *	103 *	65 *	0.006 *	6695 *	0.019
*135 *	911756 *	6 *	49.973 *	5470536 *	15.520
*136 *	1 *	73 *	0.000 *	73 *	0.000
*137 *	998 *	43 *	0.055 *	42914 *	0.122
*138 *	1 *	12 *	0.000 *	12 *	0.000

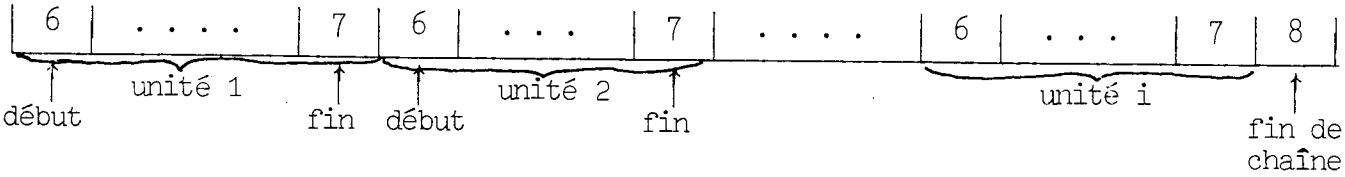
III.4 ANALYSE DES RESULTATS

La compréhension du tableau de chiffres obtenu exige la connaissance dans le détail de toutes les particularités du simulateur. Il n'est donc pas possible ici de commenter tous les aspects des résultats contenus dans les trois pages précédentes. Nous essayerons d'en dégager les principales conclusions et renvoyons le lecteur à la documentation interne sur la compilation et la simulation des langages CASSANDRE et LASCAR pour de plus amples informations.

a) Tours de stabilisation

Le nombre moyen de tours de stabilisation découle du nombre d'appels des routines n° 7 et 8. A ce propos, quelques explications supplémentaires sont nécessaires.

La chaîne de simulation est découpée en sous-chaînes, qui dirigent l'interprétation des instructions des unités de la description. Chaque sous-chaîne se termine par le code 7, qui appelle la routine de fin d'unité. La chaîne elle-même se termine par le code 8, qui appelle la routine de fin de chaîne. C'est cette deuxième routine qui, lors des tours en S, teste les indicateurs de modification. Si une valeur de signal au moins a été modifiée, le pointeur de chaîne est ré-initialisé au début pour une nouvelle exécution.



Les mesures correspondent à 1000 cycles de calcul. Il y a donc eu 1000 appels de la routine n° 8 pour les tours en R, et 1001 appels de cette routine pour la vérification de stabilité lors des tours en S. Les 1801 autres appels ont donc eu lieu lors des tours en S de stabilisation proprement dite, ce qui donne une moyenne de 1,8 tour de stabilisation par cycle. Compte tenu de la propagation des valeurs de signaux de connexion d'unités ce chiffre de 1,8 tour de stabilisation par cycle est sans doute un minimum incompressible pour ce type de modèle. En effet, la modification d'une valeur en sortie d'une unité doit être répercutée dans l'unité englobante, et en entrée d'une autre unité s'il s'agit d'un signal connectant deux unités de même niveau d'imbrication, ce qui est toujours le cas dans notre description.

Examinons à présent les appels de la routine n° 7. La description contient cinq unités : MODELE, PMEM, PCACHE et MICROP en deux exemplaires. Un mécanisme, hérité de CASSANDRE, permet de sauter, lors des tours de stabilisation, le parcours de la sous-chaîne relative à une unité, lorsque celle-ci a atteint sa stabilité au tour précédent, et qu'aucun signal d'entrée n'a été modifié. Pour les 1000 cycles simulés, il y a donc eu 5000 appels de la routine de fin d'unité pour les tours en R et 5005 appels pour la vérification de stabilité lors des tours en S. Il reste, sur les 17 368 appels, 7 363 appels pour la stabilisation effective, d'où une moyenne de 1,47 tour de stabilisation par cycle et par unité. C'est la première fois qu'est mise en évidence l'efficacité de ce mécanisme d'optimisation [A3, pp 29 et 30].

b) Les routines les plus fréquemment appelées.

Comme il fallait s'y attendre, c'est le décodeur qui avec 49,97 % des appels, est la routine le plus fréquemment exécutée. En effet, toutes les routines, exceptée la routine CYCLE, sont appelées par le décodeur. Le décodeur est une routine très courte, puisqu'elle n'exécute que 6 instructions. Mais, en nombre total d'instructions exécutées dans l'interpréteur de simulation, elle représente 15,5 % du temps. Une première optimisation consiste donc à remplacer la chaîne interprétable par une suite d'instructions de branchements aux routines appropriées, ce qui aurait pour effet :

- de libérer un registre (en permanence, un registre sert de pointeur de chaîne, un autre de registre de base de la routine de décodage),
- de faire gagner 10 % du temps passé dans l'interpréteur de simulation (le gain serait un peu inférieur par rapport au temps total de simulation).

Une autre routine très courte (4 instructions) se détache des autres : c'est la routine n° 25, qui empile une valeur immédiate sur la pile de demi-mots. Elle est utilisée notamment pour la construction sur la pile des descripteurs de sous tableaux, et pour la construction des constantes booléennes. Cependant, avec 9 % des appels, elle n'est responsable que de 1,87 % des instructions exécutées dans l'interpréteur. Cette routine, tout à fait indispensable dans un langage interprété, se révèle moins coûteuse que nous le pensions.

Six autres routines ont une fréquence d'appel supérieure à 2 %. Parmi elles, les routines n° 1 et 2 servent à sauter des codes dans la chaîne, pour n'exécuter, selon que l'on se trouve dans un tour en R ou un tour en S, que les instructions de branchement ou au contraire que les affectations et les appels de procédures Fortran et Assembleur. Nous avions envisagé de dédoubler la chaîne, afin d'avoir une chaîne pour les instructions à exécuter lors des tours en R, et une autre chaîne pour les tours en S, ce qui aurait éliminé totalement les appels aux routines BRANCHR (n° 1) et BRANCHS (n° 2). Cependant ce travail aurait nécessité une révision complète du compilateur, pour un gain de temps de 1,6 %. Cette idée a donc été abandonnée. Il est vrai que, en écrivant notre description, nous avions à l'esprit le code généré par le compilateur, et avions pris soin de ne pas intercaler inutilement les instructions de branchement et les instructions d'affectation. Pour d'autres modèles, le coût relatif de ces deux routines aurait peut-être été plus fort. Cependant, tout utilisateur averti et soucieux de performances est capable de regrouper les instructions de son programme de la même manière, ce qui de plus en accroît la lisibilité.

La routine EMPILIU (n° 29) qui, avec 3 % des appels est responsable de 7,2 % des instructions exécutées mérite quelques commentaires.

Cette routine empile, sur la pile de demi-mots, un signal ou une horloge en mode total, c'est-à-dire tout entier, descripteur compris. Contrairement à celles dont nous avons déjà parlé, cette routine déroule un nombre d'instructions qui varie en fonction du nombre d'éléments de son paramètre. Cependant, nous nous sommes rendus compte qu'une part importante des appels concernaient un paramètre de 1 bit, en vue du test de la valeur de celui-ci (instruction si non vectorielle et test de la condition horloge). En modifiant le compilateur pour qu'il soit capable de reconnaître une expression booléenne réduite à une variable scalaire, et en introduisant une nouvelle routine qui remplace dans ce cas EMPILIU suivi de BRANCHC, on pourrait gagner 2 % ou 3 % (dans les cas favorables) du nombre d'instructions exécutées dans l'interpréteur, ce qui est une très faible amélioration, mais qui est facile à réaliser.

En résumé, les 8 routines les plus fréquemment appelées totalisent 76,7 % des appels de routines, et 28,3 % des instructions exécutées dans l'interpréteur de simulation. Si l'on exclue le décodeur, les 7 autres routines représentent 26,7 % des appels et 12,8 % des instructions exécutées. Sur ces routines, une modification raisonnable du compilateur et de la chaîne codée permettrait un gain de performance de 17 % à 18 % du nombre d'instructions.

c) Les routines longues

Nous rangeons dans cette rubrique les routines qui, pour 1 appel, déroulent plus de 100 instructions. Parmi celles qui ont été appelées pour la simulation de notre description, elles sont au nombre de 10. La longueur de ces routines est dûe à la structure de données qui a été choisie pour les variables booléennes dans CASSANDRE.

Dans le langage CASSANDRE, la numérotation des vecteurs de bit est définie, pour ce qui est des faibles et forts poids, dans l'ordre inverse des conventions du système 360. Pour une utilisation du simulateur en pas à pas, où les ordres de chargement et d'impression de variables sont fréquents, il était naturel que l'ordre de rangement des bits reflètât la définition du langage. Par contre, l'interprétation des opérateurs de conversion oblige à inverser l'ordonnancement des bits d'une variable booléenne pour obtenir une variable entière, ce qui est très coûteux. D'où la complexité des routines CVALNUM, VALBIN et VALBINP.

Un autre aspect du même problème est l'obligation, pour les variables booléennes, d'associer à chaque bit de valeur un indicateur appelé bit de présence, qui permet de savoir si le bit de valeur qu'il accompagne a déjà été chargé pendant le tour (en R ou en S) en cours d'exécution, de manière à détecter les chargements conflictuels. Dans le système CASSANDRE, il avait été choisi d'intercaler bits de valeur et bits de présence, chaque paire se trouvant côte à côte. De plus, pour des raisons d'économie en taille mémoire, dans une variable booléenne de type tableau, tous les vecteurs sont concaténés. D'où des calculs très compliqués pour empiler (routines EMPIP n° 30, EMPMP n°32) ou ranger (routines RGTIP n° 37, RGTMP n° 39) des sous-tableaux, interpréter l'opérateur de réduction (routines n° 56 à 63), et la nécessité d'employer des masques pour enlever les bits de présence en fin de tour en R ou en S (routines n° 84 à 86).

Nous n'avons pas voulu, en écrivant le premier prototype du simulateur LASCAR, ré-écrire tout ce qui fonctionnait déjà pour CASSANDRE, en l'absence de toute mesure de performance sérieuse. Nous savons à présent que les 10 routines longues, qui correspondent à 2,7 % des appels, exécutent 46,4 % des instructions de l'interpréteur de simulation. Il est clair que, dans une description écrite en LASCAR, le nombre de variables booléennes n'est pas très élevé. Dans ce cas, une meilleure performance du simulateur doit avoir la priorité sur une économie de taille mémoire occupée par le modèle. Il est donc tout à fait indispensable de revoir la structure de données des variables booléennes, notamment en :

- séparant les bits de présence des bits de valeur,
- rangeant les bits selon l'ordonnancement défini sur la machine hôte,
- cadrant les vecteurs de bits sur des frontières d'unités de mémorisation de la machine hôte.

Une telle modification ne peut pas être évitée si l'on veut doubler la performance du simulateur. Mais elle représente un travail considérable : la ré-écriture d'une partie de l'assembleur, et de plus de la moitie du simulateur.

d) Coût de l'interprétation

CASSANDRE est un langage interprété. La seule solution qui nous permettait d'obtenir rapidement un simulateur prototype du langage LASCAR compatible avec le langage CASSANDRE était de ré-utiliser au maximum tout ce qui existait déjà. De plus, un code interprétable est plus aisé à concevoir et à mettre au point qu'un code exécutable. Ces deux raisons éclairent notre choix d'interpréter LASCAR. A ce propos, nous avons été surpris de constater la difficulté que représente l'extension d'un ensemble de programmes de traitement : nous avons consacré, pendant l'écriture du système LASCAR, presque la moitié de nos efforts à comprendre le système CASSANDRE dans le détail. Et chaque fois que nous pensions pouvoir nous abstenir de "décortiquer" un module de fonctions, que ce soit dans le vérificateur, le compilateur, l'assembleur ou le simulateur, cette négligence a entraîné des erreurs très difficiles à localiser.

Il est évidemment très difficile de chiffrer à priori le gain que l'on obtiendrait en réalisant une version compilée du langage LASCAR.

Quelques estimations peuvent cependant nous donner des éléments d'appréciation. Nous raisonnerons sur des exemples simples.

• Le test de la condition horloge, c'est-à-dire de l'entité syntaxique <H> dans le cas le plus simple, est traduite dans la chaîne interprétable par l'appel aux deux routines EMPILIU et BRANCHC ; ce qui, compte tenu de deux appels du décodeur, implique l'exécution de 57 ou 59 instructions, selon que l'horloge a pour valeur 1 ou 0.

L'équivalent compilé ne comporterait que les cinq instructions suivantes (code 360) :

```
l     R1, adtablei        charge adresse tablei
la    R1, depl H (R1)     charge adresse H
lh    R1, 0 (R1)          charge H
cli   R1, 0               test H
be    ...
```

D'où un rapport de 1 à 11,6 en moyenne. Dans notre modèle, ce test est effectué 12 fois par cycle. Cet exemple, à lui seul, donnerait sur une simulation de 1000 cycles, une économie de 636 000 instructions.

• La construction, sur la pile, du descripteur d'un sous-tableau à empiler ou à dépiler est une opération très fréquente. Le nombre de codes de routines générés pour ce faire, dans la chaîne interprétable, varie en fonction du type des bornes du sous-tableau (constante ou variable) et du nombre de dimensions du tableau. Nous avons évalué les solutions compilées et interprétées pour différentes configurations. Le plus petit écart relevé est de 46 instructions. La construction d'un descripteur de sous-tableaux est effectuée avant chaque appel des routines EMPIP, EMPMP, EMPILEP, RGTIP, RGTMP, RANGEP. Dans la simulation de la molécule, la somme des appels à ces routines s'élève à 33 913. Le gain obtenu, en compilant cette opération serait, en se basant sur l'écart de 46 instructions de 4,2 %, ce qui est en dessous de la vérité.

De nombreux autres exemples peuvent être trouvés, dont il serait fastidieux de faire l'énumération. Un autre point doit cependant être soulevé. Dans une version compilée du langage LASCAR, une grande partie du travail ne serait plus effectué sur la pile, mais directement par des opérations entre registres, ce qui diminuerait considérablement les transferts d'informations entre la pile et les tables, d'où un gain de temps très important à la simulation.

Il faut toutefois souligner que la génération d'instructions exécutables serait beaucoup plus coûteuse en occupation mémoire que la génération d'un code interprétable. Compte-tenu de la taille actuelle de la chaîne de simulation produite pour un gros modèle (il est courant qu'elle dépasse 10 K octets), une version compilée du langage LASCAR ne pourrait être installée que sur les très gros systèmes, ce qui limiterait le nombre de ses utilisateurs potentiels.

CONCLUSIONS

Lorsque nous avons décidé de prélever des mesures sur l'interpréteur de simulation, nous avions l'espoir de valider quelques projets de modifications ponctuelles qui devaient, selon nos hypothèses, conduire à un gain substantiel de performances à la simulation. Cet espoir a été déçu. Par contre, nous avons acquis une bien meilleure compréhension du comportement du simulateur, et une expérience plus approfondie des mécanismes de compilation et de simulation, par la réflexion supplémentaire que l'analyse des résultats obtenus a nécessité. Nous avons retiré de ce travail deux conclusions essentielles.

Tout d'abord, la compatibilité avec CASSANDRE est extrêmement pénalisante. En effet, les routines les plus coûteuses sont celles qui s'appliquent à des variables booléennes, et qui doivent assurer :

- qu'une même variable (ou sous-variable) n'a reçu qu'une seule valeur pendant un tour de calcul,
- que les valeurs des signaux de sortie d'une unité sont répercutées sur les signaux de l'unité englobante,
- que le calcul des registre est collatéral.

De plus, le problème de la stabilisation des signaux ne peut être résolu sans imposer des restrictions au langage, ce qui implique, pour chaque cycle, plusieurs parcours de la chaîne, dont le dernier (tour de vérification) est totalement improductif.

La deuxième conclusion est qu'il est possible de gagner un facteur 2 ou 3 en temps de simulation, à condition de ré-écrire entièrement le compilateur, l'assembleur et le simulateur. Il faudrait alors revoir la structure de données des variables booléennes, et supprimer le décodeur.

Une solution intermédiaire entre la compilation et l'interprétation pourrait être envisagée. Ainsi, on pourrait produire du code exécutable pour remplacer certains groupes d'appels à des routines courtes, et des instructions de branchement aux routines longues.

Bien entendu, toute la logique du code généré devrait alors être repensée.

Une telle ré-écriture de la plus grande partie des programmes de traitement du langage LASCAR représente un très gros travail, de l'ordre de 4 à 5 homme-années pour des personnes déjà familiarisées avec les algorithmes de simulation propres à ce langage et les techniques de compilation.

Mais ici s'arrête la recherche... et commence le développement.

CONCLUSION

L'objectif que nous nous étions fixés, en définissant et mettant en oeuvre le langage LASCAR, était de mettre à la disposition des concepteurs de systèmes informatiques un outil qui, couplé avec le langage CASSANDRE, devait former un ensemble cohérent et hiérarchisé de langages de description et de simulation. De ce point de vue, il est utile de revenir sur les propriétés que nous avions énumérées comme importantes pour un tel ensemble de langages d'aide à la conception, dans l'introduction de ce mémoire.

NIVEAUX DE DESCRIPTION

Les publications qui sont parues sur les applications du langage CASSANDRE ont montré son utilité pour la description structurelle des machines, et la simulation des fonctions booléennes et des micro-programmes. Avec l'introduction des variables entières, des opérateurs arithmétiques, des procédures et de la séquentialité, le langage LASCAR permet des descriptions purement algorithmiques. Nous avons donc réalisé un ensemble permettant l'écriture de modèles à des niveaux de détails très différents.

POUVOIR DESCRIPTIF

Le pouvoir descriptif de cet ensemble de langages est, au niveau le plus fin, celui de CASSANDRE et de son "compilateur de hardware". Nous ne reviendrons pas dessus.

MODULARITE

Cette propriété est elle aussi héritée de CASSANDRE, et nous avons pris soin de la conserver. A la notion de bibliothèque d'unités syntaxiquement correctes, nous avons ajouté celle de bibliothèque de sous-programmes précompilés. L'utilisateur peut enrichir progressivement l'une et l'autre au cours de l'élaboration d'un projet.

VERIFICATION DE L'EQUIVALENCE DE DEUX MODELES

Nous avons montré qu'il était important de pouvoir, dans un modèle, remplacer un module par un autre module fonctionnellement équivalent, mais décrit à un niveau soit plus fin, soit plus abstrait. Nous avons insisté sur la nécessité de vérifier leur équivalence au préalable, et avons indiqué une méthode pour procéder à cette vérification.

GESTION DU PARALLELISME

L'algorithme de simulation du langage LASCAR est déduit de celui du langage CASSANDRE, pour lequel la simulation du parallélisme est la tâche essentielle. Nous en avons évoqué certains aspects dans la troisième partie de cet ouvrage.

Dans un programme écrit en CASSANDRE, toutes les actions élémentaires sont effectuées de manière collatéralle. Nous avons vu que, pour une description algorithmique, une telle quantité de parallélisme a pour inconvénient d'obliger l'utilisateur à un découpage trop fin du temps, découpage qui n'est plus justifié par la nécessité de représenter de manière exacte le fonctionnement du matériel. Nous avons donc introduit, dans LASCAR, la séquentialité des opérations qui sont associées à un niveau plus abstrait de description, à l'intérieur d'une même unité. Toutefois, le parallélisme des unités reste total, et est, comme dans CASSANDRE, géré automatiquement par le simulateur.

EFFICACITE

Il est bien connu que l'inconvénient majeur du langage CASSANDRE est sa lenteur à la simulation. Avec le langage LASCAR, le temps de simulation est divisé par un facteur qui, selon les jeux d'essai, se situe entre 50 et 150. L'analyse de mesures nous a cependant convaincus qu'il était possible, au prix d'une ré-écriture de la plus grosse partie des programmes de traitement, de gagner encore un facteur 2 ou 3 en performance.

SIMULATEURS CONVERSATIONNELS ET EN TRAITEMENT PAR LOTS

Il est extrêmement précieux de disposer d'un simulateur conversationnel pour mettre au point ou modifier rapidement un modèle, et beaucoup plus économique de faire exécuter de longues simulations sous un système en traitement par lots. Suivant en cela ce qui avait été réalisé pour le langage CASSANDRE, nous avons écrit pour le langage LASCAR un simulateur conversationnel sous CP/CMS, et un simulateur en traitement par lots sous OS/360. Pour répondre aux besoins d'un autre utilisateur, nous avons été amenés à aménager ces deux simulateurs pour les adapter aux systèmes VM et OS 370.

UTILITE DU "NIVEAU LASCAR"

La modélisation du multi-processeur qui a été présentée dans la deuxième partie de ce mémoire était une étape tout à fait nécessaire dans l'élaboration du projet. Une fois spécifiées les primitives matérielles et logicielles du système étudié, définie (mais non vérifiée en détails) la structure interne du processeur et choisie la technologie dans laquelle devaient être fabriqués les composants, il fallait procéder à des simulations fines pour

- savoir combien de processeurs il était envisageable de connecter à un même cache,
- connaitre le coût des mécanismes qui assurent la cohérence des informations dans le système,
- obtenir une première évaluation de la puissance de la machine.

Restreignant, dans un premier temps, notre modèle à une "molécule" du système, nous avons pu simuler en LASCAR de manière très précise le fonctionnement du cache, notamment l'algorithme d'anticipation de la prise en compte des demandes de lecture, et le parallélisme des unités composant la molécule (cache, processeurs, mémoire principale). L'obtention d'un tel niveau de finesse et de précision aurait été difficile dans un langage général de simulation. Ces expériences de simulation ont été conduites sur une trace de plusieurs milliers d'instructions, en prenant comme unité de temps la période de l'horloge mère, ce qui eut été totalement impossible, parce que trop coûteux, dans un langage de description de matériel.

Grâce à ces simulations, nous avons montré

- qu'il n'est pas possible de mettre plus de huit processeurs dans une même molécule, et que le nombre qui doit être choisi se situe entre quatre et huit
- qu'une molécule d'au moins quatre processeurs représente déjà une forte puissance de calcul, et est responsable d'un important taux d'occupation de la mémoire principale ; il faut donc prévoir une mémoire principale découpée en bancs, avec accès indépendants aux différents bancs.
- que l'invalidation des données globales d'un processus à chaque accès à une variable partagée est un mécanisme extrêmement pénalisant lorsque le cache contient un grand nombre de lignes ; l'organisation des blocs du cache doit donc être révisée, ou bien ce sont les primitives qui assurent la cohérence des informations qui doivent être repensées.

Ce type de simulation, dont les résultats permettent d'évaluer à la fois l'architecture et certaines primitives d'un système informatique, fait ainsi partie intégrante du processus de conception d'un tel système. L'écriture et la mise au point du modèle (y compris toutes les procédures de génération sur des fichiers à accès direct, de lecture et de pré-traitement de la trace d'instructions, qui ont été réalisées en assembleur), les simulations et l'analyse des résultats représentent un travail de trois mois. C'est dire la facilité d'utilisation du langage LASCAR, qui à nos yeux justifie la mise en oeuvre d'"encore un nouveau langage", mais dans un domaine où le besoin d'un outil bien adapté n'avait pas, à l'époque où nous avons amorcé cette étude, reçu de réponse.

L'ensemble des programmes de traitement du langage LASCAR que nous avons réalisé, et qui n'est, rappelons-le, qu'un prototype, répond donc de manière à peu près satisfaisante aux buts que nous nous étions fixés. Lorsque nous avons abordé ce travail, nous n'avions connaissance d'aucun autre langage ayant pour principale caractéristique de permettre des modélisations de systèmes informatiques à différents niveaux de détail. Depuis, plusieurs études ont été publiées à ce sujet. Nous avons déjà mentionné le langage DIGITEST II [A30] et le langage conçu par Monsieur SIFAKIS [A27] qui, en plus de l'accent mis sur la séparation de la partie opérative et de la partie contrôle, autorisent des descriptions structurelles aussi bien que fonctionnelles. Nous ne connaissons cependant pas, au jour où nous écrivons ces lignes, l'état d'avancement du simulateur de DIGITEST II, et savons qu'un premier prototype de l'autre langage est sur le point d'être achevé. Nous pouvons évoquer aussi le langage LALSD [A37] du Professeur SU, qui est basé sur PL/1 et dont l'objectif est de permettre des descriptions à différents niveaux de détails. Ce langage n'a malheureusement jamais été entièrement compilé.

Comparé à ces langages, LASCAR n'offre aucune construction syntaxique particulièrement élégante ou originale. Son seul mérite est que des programmes de traitement opérationnels aient été mis en oeuvre, et que des utilisateurs puissent à présent s'en servir... et le critiquer. Nous leur laisserons donc la parole, pour ce qui est du langage LASCAR.

Toutefois, nous n'estimons pas qu'avec l'ensemble CASSANDRE-LASCAR, un outil complet d'aide à la conception des systèmes informatiques ait été achevé. Il manque un maillon de la chaîne, intermédiaire entre LASCAR et les langages de type SIMULA, pour la modélisation des fonctions du logiciel de base.

Ce nouvel élément de l'outil total que nous entrevoyons devra comporter des types de variables plus sophistiqués et des primitives de prises de mesures et de statistiques, l'interface des "unités" devra être redéfini, de nouvelles primitives de contrôle et de synchronisation seront introduites, et l'asynchronisme sera la règle. Il faudra bien évidemment obtenir de meilleures performances à la simulation, tout en sachant préserver un certain type de compatibilité avec LASCAR. C'est sur ce nouveau projet que se portent à présent nos efforts.

B I B L I O G R A P H I E

A - LANGAGES DE DESCRIPTION

|1| ANCEAU F., LIDDELL P., MERMET J., PAYAN C.
"A Language to Describe Digital Systems,Application to Logic Design"
C.O.I.N.S., Miami, Decembre 1969

|2| ANCEAU F., DOUSSY J., LIDDELL P., MERMET J.,PAYAN C.
"CASSANDRE - Langage et systeme"
Seminaire de Programmation, IMAG, Mai 1970

|3| ANCEAU F., COUTURIER, DOUSSY J., PERRON F.
"Compilation et simulation du langage CASSANDRE"
Rapport final du contrat CRI "Industrialisation de CASSANDRE", Chapitre II

|4| BARBACCI M. R.
"A User's Guide To The ISPL Compiler"
Rapport interne, Carnegie-Mellon University,
Department of Computer Science, June 26, 1975

|5| BECKER M., KLAR R., SPIES P.P.
"The Erlangen Computer Design Language ERES"
Workshop on Computer Hardware Description Languages, Darmstadt, 31 Juillet - 2 Aout 1974

|6| BELL C. G. et NEWELL A.
"The PMS and ISP Descriptive System for Computer Structures"
AFIPS Conference Proceedings, Spring Joint Computer Conference 1970, Vol. 36

|7| BELL C.G. et NEWELL A.
"Computer Structures: Readings and Examples"
McGraw-Hill, New York, 1971

|8| BORRIONE D.
"LASCAR : a LAnguage for Simulation of Computer ARchitecture"
International Symposium on Computer Hardware Description Languages and
Their Applications, New York, 3-5 Septembre 1975

|9| BRESSY Y., DAVID B., FANTINO Y., MERMET J.
"A Hardware Compiler for Interactive Realization of Logical Systems Described in CASSANDRE"
International Symposium on Computer Hardware Description Languages and
Their Applications, New York, 3-5 Septembre 1975

|10| CHU Y.
"An Algol-Like Computer Design Language"
CACM, Vol. 8, No. 10, Octobre 1965

|11| CHU Y.
"A Computer Design Language"
Ecole d'Ete de l'OTAN, Hyeres, Aout 1969

|12| CHU Y.

|13| CHU Y. et MESZTENYI C. K.
"Macro Logic Design of Digital Computers"
Technical Report 67-57, Computer Science Center, University of Maryland, Novembre 1967

|14| DIETMEYER D. L.
"Introducing DDL"
Computer, Decembre 1974

|15| DULEY J. R. et DIETMEYER D. L.
"A Digital System Design Language"
IEEE Trans. on Computers, Vol. C17, No. 9, Septembre 1968

|16| DULEY J. R., DIETMEYER D. L.
"Translation of a DDL Digital System Specification to Boolean Equations"
IEEE Trans, Vol. C18, No. 4, Avril 1969

|17| GORMAN D. F. et ANDERSON J. P.
"A Logic Design Translator"
AFIPS Conference Proceedings, Fall Joint Computer Conference 1962

|18| HILL F.
"Updating AHPL"
International Symposium on Computer Hardware Description Languages and
Their Applications, New York, 3-5 Septembre 1975

|19| HOFFMANN R.
"The Hardware Description and Programming Language HDL "
Bericht Nr. 75-04, Technische Universitat Berlin, Mars 1975

|20| KNUDSEN M. J.
"PMSL, An Interactive Language for System-Level Description and Analysis
of Computer Structures"
PhD Thesis, Carnegie-Mellon University, Department of Computer Science, Avril 1973

|21| LUND J.
"LOGAL - Logic Algorithmic Language"
Univac Technical Memo A00317, Roseville, Minnesota, 5 Mars 1973

|22| MARCZYNSKI R., PULCZYN W., SOCHACKI J.
"OSM - Microprogrammed Hardware Structure Description Language"
International Symposium on Computer Hardware Description Languages
and their Applications, New York, 3-5 Septembre 1975

|23| McKAY A. R.
"Comment on Computer-Aided Design: Simulation for Digital Design Logic"
IEEE Trans. on Computers, Vol. C18, Septembre 1969

|24| MACLURE R. M.
"A Programming Language for Simulating Digital Systems"
JACM, Vol. 12, Janvier 1965

|25| MERMET J.
"Etude Methodologique de la Conception Assistee par Ordinateur des Systemes Logiques: CASSANDRE"
These de Doctorat d'Etat, Universite de Grenoble, Avril 1973

|26| MERMET J.
"Definition du Langage CASSANDRE"
These de Docteur Ingenieur, Universite de Grenoble, Mars 1970

|27| MOALLA M., SIFAKIS J., ZACHARIADES M.
"Un Langage d'Aide a la Conception et a la Simulation de Systemes Complexes"
Rapport de Recherche ENSIMAG, No 16, Octobre 1975

|28| PARNAS D. L.
"A Language for Describing the Functions of Synchronous Systems"
CACM, Vol. 9, No. 2, Fevrier 1966

|29| PILOTY R.
"Functional and Structural Segmentation in RTS III, a Register Transfer Language"
Workshop on Computer Hardware Description Languages, Darmstadt, 31 Juillet - 2 Aout 1974

|30| RAMMIG F. J.
"DIGITEST II: An Integrated Structural and Behavioral Language"
International Symposium on Computer Hardware Description Languages and
Their Applications, New York, 3-5 Septembre 1975

|31| SCHEFF B. H. et YOUNG S. P.
"Gate-Level Logic Simulation"
Design Automation of Digital Systems, M. A. Breuer ed., Prentice Hall, Englewood Cliffs,
N. J. 1972

|32| SCHLAEPPI H. P.
"A Formal Language for Describing Machine Logic, Timing and Sequencing (LOTIS)"
IEEE Trans. on Electronic Computers, Vol. EC, Aout 1964

|33| SCHORR H.
"Computer-Aided Digital System Design and Analysis Using a Register Transfer Language"
IEEE Trans. on Electronic Computers, Vol. Ec-13, No. 6, Decembre 1964

|34| SIEWIOREK D.
"Introducing ISP"
"Introducing PMS"
Computer, Decembre 1974

|35| SRINIVASAN C. V.
"An Introduction to CDL1, a Computer Description Language"
Scientific Report No. 1 AFCRL-67-0565, Air Force Cambridge Research Laboratories,
Bedford, Mass., Septembre 1967

|36| SRINIVASAN C. V.
"Formal Definition of CDL1, a Computer Description Language"
Scientific Report No. 2, AFCRL-67-0588, Air Force Cambridge
Research Laboratories,
Bedford, Mass., Octobre 1967

|37| SU S. Y. H., Baray M. B.
"LALSD - A Language for Automated Logic and System Design"
Proceedings of International Computer Symposium 1975

B - LANGAGES GENERAUX DE SIMULATION

|1| ANDRE J.
"Introduction a Simula .67"
Seminaire de programmation de l'Universite de Toulouse,
Janvier 1971

|2| BIRTWISTLE G.
"Notes on the Simula Language"
Norwegian Computing Centre, Publication No. 5-7, Avril 1969

|3| BIRTWISTLE G.
"Notes on the Simula System Classes"
Norwegian Computing Centre, Publication No. 5-8, Avril 1969

|4| BUXTON J. N. et LASKI J. G.
"Control and Simulation Language"
Computer Journal, Vol. 5, No. 3, Octobre 1962

|5| CII
"Simula sous Siris7/Siris8"
Manuel d'Utilisation, ref. 4179E/FR, Novembre 1972

|6| DAHL O. J. et NYGAARD K.
"SIMULA: an Algol-based Simulation Language"
CACM, Vol. 9, September 1966

|7| EFRAN R. et GORDON G.
"A General Purpose Digital Simulator and Examples of its
Application, Part I,
Description of the Simulator"
IBM Systems Journal, Vol. 3, No. 1, 1964

|8| GORDON G.
"A General Purpose Systems Simulation Program"
AFIPS Conference Proceedings, Vol. 20, Decembre 1961

|9| GORDON G.
"A General Purpose System Simulator"
IBM Systems Journal, Vol. 1, Septembre 1962

|10| HERSCOVITCH R. et SCHNEIDER T. H.
"GPSSIII: an Expanded General Purpose Simulator"
IBM Systems Journal, Vol. 4, No. 3 1965

|11| KIVIAT P. J., COLKER A.
"GASP- A General Simulation Program"
Rand Corp., P-2864, Santa Monica, 1964

|12| KIVIAT P. J., VILLAVUEVA R. E., MARKOWITZ H. M.
"The Simscript II Programming Language"
Prentice Hall, Englewood Cliffs, N. J. 1969

|13| KNUTH D. E. et McNELEY J. L.
"SOL - A Symbolic Language for General Purpose Systems
Simulation"

IEEE Trans. on Electronic Computers, Vol EC-13, No. 4, Aout 1964

|14| KNUTH D. E. et McNELEY J. L.
"A Formal Definition of SOL"
IEEE Trans. on Electronic Computers, Vol. EC-13, No. 4, Aout 1964

|15| MARKOWITZ H. M., HAUSNER B., KAU H. W.
"Simscript: A Simulation Programming Language"
Prentice Hall, Englewood Cliffs, N. J., 1963

|16| NIELSEN N. R.
"ECSS: an Extendable Computer System Simulator"
Proceedings of the Third Conference on Applications of Simulation, Los Angeles, 1969

|17| NYGAARD K.
"SIMULA: an Extension of Algol to the Description of Descrete Event Networks"
Proceedings IFIP Congress 1962, North Holland Publishing Co., Amersterdam

|18| PETRONE L.
"On a Simulation Language Completely Defined on to the Programming Language PL/1"
Simulation Programming Languages, Buxton ed, North Holland 1968

|19| REDDY Y. V. and BRYAN R. H.
"DESPL/1: A discrete Simulation Language Based on PL/1"
Summer Computer Simulation Conference, Juillet 1973, Montreal, Canada

|20| RETTENMAYER J. W.
"SIMPL/1: A Simulation Programming Language"
Winter Simulation Conference, Janvier 1974

|21| SCHERER C.
"Simula par l'exemple"
Publication CII, ref. 5224T/FR, decembre 1972

C - MODELES THEORIQUES

|1| BELADY L. A.
"A Study of Replacement Algorithms for a Virtual Storage Computer"
IBM Syst. Journal, Vol. 5, No. 2, 1966

|2| BRADSHAW F. T.
"Directed Graph Models for Hardware/Software Design"
International Symposium on Computer Hardware Description Languages and
Their Applications, New York, 3-5 Septembre 1975

|3| BUZEN J.
"Analysis of System Bottlenecks Using a Queueing Network Model"
ACM, SIGOPS Workshop on System Performance Evaluation, Avril 1971

|4| COFFMAN E. G.
"Analysis of a Drum Input/Output Queue Under Scheduled Operations in a
Paged Computer System"
JACM, Vol. 16, No. 1, Janvier 1969

|5| COMMONER R., HOLT A. W., EVEN S., PNUELI A.
"Marked Directed Graphs"
Journal of Computer and System Sciences, Vol. 5, 1971, pp 511-523

|6| DENNING P. J. et SCHWARTZ S. C.
"Properties of the Working Set Model"
CACM, Vol. 15, No. 3, Mars 1972

|7| DENNING P.J.
"The Working Set Model for Program Behavior"
CACM, Vol. 11, No. 5, Mai 1968

|8| DENNING P.J.
"A Statistical Model for Console Behavior in Multiuser Computers "
CACM, Vol. 11, No. 9, Septembre 1968

|9| ESTRIN G. et KLEINROCK L.
"Measures, Models and Measurements for Time- Shared Computer Utilities"
Proceedings ACM National Meeting, 1967

|10| ESTRIN G., TURN R.
"Automatic Assignment of Computations in a Variable Structure Computer System "
IEEE Trans. on Electronic Computers, EC12, Decembre 1963, pp 756-773

|11| FOO S. Y., MUSGRAVE G.
"Comparison of Graph Models for Parallel Computation and their Extension"

International Symposium on Computer Hardware Description Languages and
Their Applications, New York, 3-5 Septembre 1975

|12| FRANK H.
"Analysis and Optimization of Disk Storage Devices for Time-Sharing Systems"
JACM, Vol. 16, No. 4, Octobre 1969

|13| GELEMBE E.
"A Unified Approach to the Evaluation of a Class of Replacement Algorithms"
IEEE Trans on Computers, Juin 1973

|14| GLASER E. L.
"Introduction and Overview of the LOGOS Project"
Digest 6th Annual IEEE Computer Society International Conference, 1972, pp 175-177

|15| GORDON W.J. et NEWELL G. F.
"Closed Queuing Systems With Exponential Servers"
Operation Research, Vol. 1, No. 2. June 1969

|16| HUGHES P. H. et MOE G.
"A Structural Approach to Computer Performance Analysis"
AFIPS Conference Proceedings, Joint Computer Conference 1973, Vol. 42

|17| KARP R. M., MILLER R. E.
"Parallel Program Schemata"
Journal of Computer and System Science, Vol. 3, 1969, pp 147-195

|18| KLEINROCK L.
"Time-Shared Systems: A Theoretical Treatment"
JACM, Vol. 14, No. 2, Avril 1967

|19| KLEINROCK L.
"A Continuum of Time-Sharing Scheduling Algorithms"
AFIPS Conference Proceedings, Spring Joint Computer Conference 1970

|20| LENFANT J.
"Evaluation sur des Modeles de Comportement de Programme de la Taille d'un
Ensemble de Travail"
Colloques IRIA, avril 1974

|21| LEROUDIER J., POTIER D.
"A Two Level Control Scheme for Multiprogrammed Virtual Memory Computer Systems"
Rapport LABORIA No. 141, Novembre 1975

|22| MUNTZ R. R., COFFMAN E. G.,
"Optimal Preemptive Scheduling on Two-Processor Systems"
IEEE Trans. on Computers, Vol. C18, No. 11, Novembre 1969

|23| PARENT M.
"Presentation of the Control Graph Models"
Actes du "Colloque International sur Les Aspects Theoriques et Pratiques des
Systemes d'Exploitation", IRIA, Avril 1974

|24| PETRI C. A.
"Kommunikation Mit Automaton"
Schriften des Reinsch - West Falishen Inst.
Instrumentelle Math. und der Universitat Bonn, No 2, Bonn, 1962

|25| POTIER D.
"La Modelisation des Systemes Informatiques"
Bulletin de Liaison de l'IRIA, Janvier 1974

|26| SASTRY K. V., KLAIN R. Y.
"On The Performance of Certain Multiprocessor Computer Organizations"
IEEE Trans. on Computers, Vol C24, No 11, November 1975

|27| THEOREY T. J. et PINKERTON T. B.
"A Comparative Analysis of Disk Scheduling Policies"
CACM, Vol. 15, No. 3, Mars 1972

|28| TURNBULL C. J. M.
"A Comparative Analysis of Several Disk Scheduling Algorithms"
Technical Report CSRG, No. 18, Septembre 1972

|29| WILLIAMS J. G.
"Asymmetric Memory Hierarchies"
CACM, Vol. 16, No. 4, Avril 1973

D - METHODOLOGIE ET APPLICATIONS DE LA SIMULATION

|1| BEILNER H., WALDBAUM G.
"Statistical Methodology For Calibrating a Trace-Driven Simulator of a Batch
Computer System"
IBM Research, RC3855, 18 Mai 1972

|2| BELL J., CASASENT D. BELL C. G.
"An Investigation of Alternative Cache Organisations"
IEEE Trans. on Computers, Vol C23, No. 4,
Avril 1974

|3| BELL T. E.
"Performance Determination: The Selection of Tools, if any"
AFIPS Conference Proceedings, Vol. 42, Juin 1973

|4| BOEHM B. W.
"Computer Systems Analysis Methodology: Studies in Measuring, Evaluating
and Simulating Computer Systems"
Rand Corporation, R-520-NASA, Septembre 1970

|5| BOWDON E. K., MAMRAK S.A., SALZ F.R.
"Simulation- A Tool for Performance Evaluation in Network Computers"
AFIPS Conference Proceedings, Vol. 42, Juin 1973, pp. 121-132

|6| BREUER M. A.
"Techniques for the Simulation of Computer Logic"
CACM, Vol. 7, Juillet 1964

|7| CALINGAERT P.
"System Performance Evaluation: Survey and Appraisal"
CACM, Vol. 10, No. 1, Janvier 1967

|8| FAGAN T. L. et WILSON M. A.
"Monte Carlo Simulation of System Reliability"
Proceedings 23rd National Conference of the ACM, 1968

|9| FUCHI K., TANAKA H., MANAGO Y., YUBA T.
"A Program Simulator by Partial Interpretation "
ACM 2nd Symposium on Operating Systems Principles,Princeton University,
Octobre 1969

|10| GARWICK J. V.
"Do We Need all these Languages"
Simulation Programming Languages, Buxton ed., North-Holland, 1968

|11| GRIMMOND R.
"An Analysis of Real and Simulated Statistics for System Design Purposes"
Computer Journal, Vol. 5, No. 2, Juillet 1962

|12| JONES M. M.
"Digital Simulation Methodology and Digital Simulation Languages"
Cours Polycopie M.I.T.

|13| LEHMAN M., ESHED R, et NETTER Z.
"The Checking of Computer Logic by Simulation on a Computer"
Computer Journal, Vol. 6, No. 2, Juillet 1963

|14| LEROUDIER J., PARENT M.
"Quelques Aspects de la Modelisation des Systemes Informatiques par Simulation a Evenements Discrets"
Note interne IRIA, 1974

|15| MACDOUGALL M.H.
"Computer System Simulation: an Introduction"
Computing Surveys, Vol. 2, No. 3, Septembre 1970

|16| MARTIN F. F.
"Computer Modeling and Simulation"
John Wiley and Sons Inc., New York, 1968

|17| NIELSEN Norman R.
"Computer Simulation of Computer System Performance"
Proceedings ACM National Meeting, 1967

|18| NIELSEN N. R.
"The Simulation of Time Sharing Systems"
CACM, Vol 10, No 7, Juillet 1967

|19| SEAMAN P. H. et SAUCY R
"Simulating Operating Systems"
IBM Systems Journal, Vol. 8, No. 4, 1969

|20| SHERMAN S., BASKET F., BROWNE J. C.
"Trace-Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System"
CACM, Vol. 15,No. 12, Decembre 1972

|21| VAN SLYKE R., CHOU W., FRANK H.
"Avoiding Simulation in Simulating Computer Communication Networks"
AFIPS Conference Proceedings, Vol. 42, Juin 1973

|22| WATSON R.
"Computer Performance Analysis : Applications of Accounting Data"
Rand Corporation, Rapport R-573-NASA/PR, Mai 1971

E - ARTICLES DE SYNTHESE

|1| BARBACCI M. R.
"A Comparison of Register Transfer Languages for Describing Computers
and Digital Systems"
IEEE Trans. on Computers, Vol. C24, No. 2, Fevrier 1975

|2| KIVIAT P. J.
"Development of Discrete Digital Simulation Languages"
Simulation fevrier 1967

|3| McKINNEY J. M.
"A Survey of Analytical Time-Sharing Models"
Computing Surveys, Vol 1, No 2, June 1969

|4| SU S. Y. H.
"A Survey of Digital Hardware Description Languages"
Workshop on Computer Hardware Description Languages, Darmstadt, 31 Juillet - 2 Aout 1974

|5| SU S. Y. H.
"A Survey of Computer Hardware Description Languages in the U. S.A."
Computer, Decembre 1974

|6| TEICHROEW D. et LUBIN J. F.
"Computer Simulation- Discussion of the Technique and Comparison of Languages"
CACM, Vol. 9, No. 10, Octobre 1966

|7| TOCHER K. D.
"Review of Simulation Languages"
Operational Research Quaterly, Vol. 16, No. 2, Juin 1965

F - DIVERS

|1| ANCEAU F., DOUSSY J., PERRON F.
"Compilation du Langage CASSANDRE"
Rapport Interne IMAG, 8 Mai 1970

|2| BAER J. L.
"A Survey of Some Theoretical Aspects of Multiprocessing"
Computing Surveys, Vol. 5, No. 1, Mars 1973

|3| BARBACCI M. R., SIEWIOREK D. P.
"Some Observations on Modular Design Technology and the Use of Microprogramming"
Rapport Technique, Carnegie-Mellon University, Juillet 1974

|4| BARBACCI M. R., SIEWIOREK D. P.
"Some Aspects of the Symbolic Manipulation of Computer Descriptions"
Workshop on Hardware Description Languages, Darmstadt, 31 Juillet-2 Aout 1974

|5| BARBACCI M. R., SIEWIOREK D. P.
"The CMU RT-CAD System : An Innovative Approach to Computer aided Design"
Rapport Technique, Carnegie-Mellon University, Juin 1975

|6| BELL T. E.
"Computer Performance Analysis : Measurement, Objectives and Tools"
Rand Corporation, Rapport R-584-NASA/PR, Fevrier 1971

|7| BORRIONE D., CENSIER, MAZARE G., RECOQUE A.
"Etude et Modelisation de Systemes Multiprocesseurs Composés de Modules Monolithiques du Type Composant"
Rapports No 1,2,3, Contrat DRME, No. 73.34.848.00.480.75.01

|8| BURROUGHS Corporation
"B8500 Simulation for B5500, Programmer's Reference Manual", 1966

|9| GECSEI J., SLUTZ D. R., TRAIGER I. L.
"Evaluation Techniques for Storage Hierarchies"
IBM Systems Journal, No. 2, 1970

|10| GUERIN M.
"Le Systeme 360 Mod. 195"
Seminaires IMAG, 19 Mars 1971

|11| IBM Corpora
"Principles of Operation System 360"

|12| IVERSON K. E.
"A Programming Language"
John Wiley and Sons, New York, 1962

|13| JACOLIN H.
"Optimisation de la Simulation en Langage CASSANDRE"
Projet de DEA, Septembre 1974

|14| KUCK D. J., and al.
"On the Number of Operations Simultaneously Executable in Fortran-like Programs, and their Resulting Speedup"
IEEE Trans on Computers, Vol C21

|15| KUCK D. J., and al
"Measurements of Parallelism in Ordinary Fortran Programs"
Computer, Janvier 1974, pp 37-45

|16| LESSER V. R.
"Dynamic Control Structures and Their Use in Emulation"
SLAC Report No. 157, Stanford, California, Octobre 1972

|17| LEVY J.
"Computing with Multiple Microprocessors"
International Workshop on Computer Architecture, Grenoble, Juin 1973

|18| LIPTAY J. S.
"Structural Aspects of the System 360 Model 85-II The Cache"
IBM Systems Journal, Vol. 7, No. 1. 1968

|19| MINSKY M. L.
"Matter, Mind and Models"
Semantic Information Processing, MIT Press, Cambridge, Massachussetts,1968,p426

|20| PAUL J. L. et TROY R.
"Le Projet Segma"
Seminaire IMAG, 29 Mai 1974

|21| PETEUL HARMEL B.
"La Mise au Point de Programmes par Simulation- Realisation d'un Support
Conversationel de Mise au Point: PILOTE"
These, Universite de Grenoble, Juin 1971

|22| TRAIGER I. L. et MATTSON R. L.
"The Evaluation and Selection of Technologies for Computer Storage Systems"
IBM Research, RJ 967, Fevrier 1972

|23| YETTER I. H.
"High Speed Fault Simulation for Univac 1107 Computer System"
Proceedings 23rd National Conference of the ACM, 1968

|24| PARHAMI B.
"Application of APL for rapid verification of a digital system architecture"
Congrès APL 75, Pise, 11 au 13 juin 1975

ANNEXES

ANNEXE I

GRAMMAIRE DU LANGAGE LASCAR

NOTATIONS

Tout programme LASCAR peut être écrit à l'aide des caractères disponibles sur un terminal classique, télétype par exemple. Les mots clè sont soulignés dans le texte : en pratique, ils sont écrits entre apostrophes dans les programmes, Algol 60.

iserons la convention d'écriture suivante :

ité syntaxique est désignée par son nom entre crochets,

e verticale sert de séparateur entre les différentes alternatives règle de grammaire.

syntaxique sera mise sous la forme normale de Backus :

< *METAVARIABLE* > ::= < *DEFINITION 1* > | < *DEFINITION 2* >

De manière à diminuer le nombre de règles, nous utiliserons pour exprimer la compatibilité de type des méta-variables d'une même règle une notation qui s'apparente aux doubles grammaires, sans toutefois reprendre tout le formalisme des doubles grammaires : un symbole de l'alphabet grec, écrit dans plusieurs éléments non terminaux d'une règle syntaxique, peut être remplacé par l'un quelconque des mots qu'il représente, mais le même pour toutes ses occurences dans cette règle.

Exemple

< *EXPRESSION* γ > ::= < *EXPRESSION CONDITIONNELLE* γ > |
< *EXPRESSION* γ *SIMPLE* >

γ → *BOOLEENNE* | *D'ETAT*

est équivalent aux règles :

< *EXPRESSION BOOLEENNE* > ::= < *EXPRESSION CONDITIONNELLE BOOLEENNE* > |
< *EXPRESSION BOOLEENNE SIMPLE* >

< *EXPRESSION D'ETAT* > ::= < *EXPRESSION CONDITIONNELLE D'ETAT* > |
< *EXPRESSION D'ETAT SIMPLE* >

Enfin, la grammaire du langage LASCAR étant un sur-ensemble de la grammaire du langage CASSANDRE, les règles spécifiques aux notions purement LASCAR seront distinguées par des caractères gras.

PROGRAMME LASCAR

< *PROGRAMME* > ::= < *ENTETE D'UNITE* > ;
< *LISTE DE DECLARATIONS ET SPECIFICATIONS* > ;
< *CORPS DE PROGRAMME* > ;

ENTETE D'UNITE

< *ENTETE D'UNITE* > ::= *unité* < *IDENTIFICATEUR* >
(< *LISTE D'ENTREE* > ; < *LISTE DE SORTIE* >) ;
< *LISTE D'ENTREE* > ::= < *IDENTIFICATEUR d'E/S* > |
< *LISTE D'ENTREE* >, < *IDENTIFICATEUR d'E/S* >
< *IDENTIFICATEUR d'E/S* > ::= < *IDENTIFICATEUR* > |
< *IDENTIFICATEUR DE TABLEAU* >
< *LISTE DE SORTIE* > ::= < *IDENTIFICATEUR d'E/S* > |
< *LISTE DE SORTIE* >, < *IDENTIFICATEUR d'E/S* >

DECLARATIONS ET SPECIFICATIONS

< *LISTE DE DECLARATIONS ET SPECIFICATIONS* > ::= < *SPECIFICATION* > |
< *DECLARATION* > |
< *LISTE DE DECLARATIONS ET SPECIFICATIONS* > ; < *SPECIFICATION* > |
< *LISTE DE DECLARATIONS ET SPECIFICATIONS* > ; < *DECLARATION* >
< *SPECIFICATION* > ::= *horlogemère* < *LISTE D'IDENTIFICATEURS d'E/S* >
< *LISTE D'IDENTIFICATEURS d'E/S* > ::= < *IDENTIFICATEUR d'E/S* > |
< *LISTE D'IDENTIFICATEURS d'E/S* >, < *IDENTIFICATEUR d'E/S* >
< *DECLARATION* > ::= < *DECLARATION DE VARIABLE* > |
< *DECLARATION DE SOUS-UNITE* > |
< DECLARATION DE PROCEDURE >

DECLARATIONS DE VARIABLES

< *DECLARATION DE VARIABLE* > ::= < DECLARATION DE VARIABLE SIMPLE > |
< DECLARATION DE TABLEAU ENTIER > |
< *DECLARATION DE VARIABLE CASSANDRE* >

< DECLARATION DE VARIABLE SIMPLE > ::= < TYPE SIMPLE >
< LISTE D'IDENTIFICATEURS >

< DECLARATION DE TABLEAU ENTIER > ::= tableau entier
< LISTE D'IDENTIFICATEURS DE TABLEAU >

< *DECLARATION DE VARIABLE CASSANDRE* > ::= < *TYPE CASSANDRE* >
< *LISTE MIXTE D'IDENTIFICATEURS* >

< TYPE SIMPLE > ::= entier | compteur

< *TYPE CASSANDRE* ::= *signal* | *registre* | *horloge* | *état*

< *LISTE D'IDENTIFICATEURS* > ::= < *IDENTIFICATEUR* > |
< *LISTE D'IDENTIFICATEURS* >, < *IDENTIFICATEUR* >

< *LISTE D'IDENTIFICATEURS DE TABLEAU* > ::= < *IDENTIFICATEUR DE TABLEAU* > |
< *LISTE D'IDENTIFICATEURS DE TABLEAU* >, < *IDENTIFICATEUR DE TABLEAU* >

< *IDENTIFICATEUR DE TABLEAU* > ::= < *IDENTIFICATEUR* >
(< *LISTE DE PAIRES DE BORNES* >)

< *LISTE DE PAIRES DE BORNES* > ::= < *PAIRE DE BORNES* > |
< *LISTE DE PAIRES DE BORNES* >, < *PAIRE DE BORNES* >

< *PAIRE DE BORNES* > ::= < *ENTIER SANS SIGNE* > : < *ENTIER SANS SIGNE* >

< *LISTE MIXTE D'IDENTIFICATEURS* > ::= < *IDENTIFICATEUR* > |
< *IDENTIFICATEUR DE TABLEAU* > |
< *LISTE MIXTE D'IDENTIFICATEURS* >, < *IDENTIFICATEUR* > |
< *LISTE MIXTE D'IDENTIFICATEURS* >, < *IDENTIFICATEUR DE TABLEAU* >

DECLARATION DE SOUS-UNITE

< *DECLARATION DE SOUS-UNITE* > ::= *externe* < *LISTE DE SOUS-UNITES* > ;

< *LISTE DE SOUS-UNITES* > ::= < *SOUS-UNITE* > |
< *LISTE DE SOUS-UNITES* >, < *SOUS-UNITE* >

< *SOUS-UNITE* > ::= < *IDENTIFICATEUR* > (< *ENTREES* > ; < *SORTIES* >)

< *ENTREES* > ::= < *ENTREES* >, *horloge* | < *ENTREES* >, | ϕ |
< *ENTREES* >, (< *LISTE DE PAIRES DE BORNES* >)

< *SORTIES* > ::= < *SORTIES* >, | < *SORTIES* >, *horloge* | ϕ |
< *SORTIES* >, (< *LISTE DE PAIRES DE BORNES* >)

ϕ désigne l'ensemble vide.

Les entrées et les sorties sont par défaut des signaux, sauf si le mot clé horloge apparaît.

Dans le cas de signaux scalaires, rien n'est indiqué : le nombre de virgules est suffisant pour les décompter. Dans le cas de signaux ou d'horloges non scalaires, il faut en indiquer les dimensions.

DECLARATIONS DE PROCEDURE

< DECLARATION DE PROCEDURE > ::= procédure < LISTE DE PROCEDURES > ; |
fortran < LISTE DE PROCEDURES > ;

< LISTE DE PROCEDURES > ::= < NOM DE PROCEDURE > |
< LISTE DE PROCEDURES >, < NOM DE PROCEDURE >

< NOM DE PROCEDURE > ::= < IDENTIFICATEUR > |
< IDENTIFICATEUR > (< ENTIER SANS SIGNE >)

CORPS DE PROGRAMME

< CORPS DE PROGRAMME > ::= < INSTRUCTIONS TOUJOURS VRAIES > | < AUTOMATE > |
< INSTRUCTIONS TOUJOURS VRAIES > ; < AUTOMATE >

< INSTRUCTIONS TOUJOURS VRAIES > ::= < LISTE D'INSTRUCTIONS >

< AUTOMATE > ::= < ETIQUETTE > : < LISTE D'INSTRUCTIONS > |
< AUTOMATE > ; < ETIQUETTE > : < LISTE D'INSTRUCTIONS >

< LISTE D'INSTRUCTIONS > ::= < LISTE DE BRANCHEMENTS > |
< INSTRUCTION D'AFFECTATION > |
< LISTE D'INSTRUCTIONS > ; < LISTE DE BRANCHEMENTS > |
< LISTE D'INSTRUCTIONS > ; < INSTRUCTION D'AFFECTATION >

< LISTE DE BRANCHEMENTS > ::= < BRANCHEMENT > |
< LISTE DE BRANCHEMENTS > ; < BRANCHEMENT >

< BRANCHEMENT > ::= < ITERATION > | < INSTRUCTION CONDITIONNELLE D'ITERATION
< INSTRUCTION CONDITIONNABLE >

< INSTRUCTION CONDITIONNABLE > ::= < INSTRUCTION FAIRE > |
< INSTRUCTION CONDITIONNELLE > |
< BRANCHEMENT SIMPLE > |
< INSTRUCTION D'AFFECTATION >

< INSTRUCTION CONDITIONNELLE > ::= < INSTRUCTION CONDITIONNELLE SIMPLE > |
< INSTRUCTION CONDITIONNELLE VECTORIELLE >

< *BRANCHEMENT SIMPLE* > ::= < *CONNEXION DE SIGNAL* > | < *CONNEXION D'HORLOGE* > |
< *CONNEXION D'UNITE* > | < *CONNEXION DE SIGNAL UNITE* >

< *INSTRUCTION D'AFFECTATION* > ::= < *CONDITION HORLOGE* >
< *LISTE D'AFFECTATIONS* >

< *LISTE D'AFFECTATIONS* > ::= < *AFFECTATION* > |
< *LISTE D'AFFECTATIONS* >, < *AFFECTATION* >

< *CONDITION HORLOGE* > ::= << *EXPRESSION D'HORLOGE* > >

< *AFFECTATION* > ::= < *AFFECTATION ITERATIVE* > |
< *AFFECTATION CONDITIONNELLE D'ITERATION* > |
< *AFFECTATION CONDITIONNABLE* >

< *AFFECTATION CONDITIONNABLE* > ::= < *AFFECTATION CONDITIONNELLE* > |
< *AFFECTATION SIMPLE* >

< *AFFECTATION CONDITIONNELLE* > ::= < *AFFECTATION CONDITIONNELLE SIMPLE* > |
< *AFFECTATION CONDITIONNELLE VECTORIELLE* >

< *AFFECTATION SIMPLE* > ::= < *CHARGEMENT DE REGISTRE* > |
< *AFFECTATION D'HORLOGE* > |
< AFFECTATION D'ENTIER > |
< APPEL DE PROCEDURE > |
< *INSTRUCTION ALLERA* >

BRANCHEMENTS SIMPLES

< *CONNEXION DE SIGNAL* > ::=
< *VARIABLE DE TYPE SIGNAL* > := < *EXPRESSION BOOLEENNE* > |
< VARIABLE DE TYPE SIGNAL > := ! (< EXPRESSION ARITHMETIQUE >)

< *CONNEXION D'HORLOGE* > ::=
< *VARIABLE DE TYPE HORLOGE* > := < *EXPRESSION D'HORLOGE* >

< *CONNEXION D'UNITE* > ::= < *IDENTIFICATEUR DE SOUS-UNITE* >
(< *ENTREES1* > ; < *SORTIES1* >) |
< *IDENTIFICATION DE SOUS-UNITE* > < *DUPLICATION* > (< *ENTREES1* > ; < *SORTIES* >)

< *ENTREES1* > ::= < *EXPRESSION D'HORLOGE* >, < *ENTREES1* > |
< *EXPRESSION BOOLEENNE* >, < *ENTREES1* > |
, < *ENTREES1* > | ϕ | < *EXPRESSION D'HORLOGE* > |
< *EXPRESSION DE SIGNAL* >

< *SORTIES1* > ::= < *VARIABLE DE TYPE SIGNAL* >, < *SORTIES1* > |
< *VARIABLE DE TYPE HORLOGE* >, < *SORTIES1* > |
, < *SORTIES1* > | ϕ | < *VARIABLE DE TYPE HORLOGE* > |
< *VARIABLE DE TYPE SIGNAL* >

< *DUPLICATION* > ::= ↑ < *ENTIER SANS SIGNE* > ↑ |
↑ < *EXPRESSION D'ITERATION* > ↑

< CONNEXION DE SIGNAL-UNITE > ::=
*< IDENTIFICATEUR DE SOUS-UNITE > (< ENTREES1 >, *,*
< ENTREES1 > ; < SORTIES1 >) := < EXP BOOL H > |
< IDENTIFICATEUR DE SOUS-UNITE > < DUPLICATION > (< ENTREES1 >,
**, < ENTREES1 > ; < SORTIES1 >) := < EXP BOOL H >*
< EXP BOOL H > ::= < EXPRESSION BOOLEENNE > | < EXPRESSION D'HORLOGE >
< SIGNAL-UNITE > ::= < IDENTIFICATEUR DE SOUS-UNITE > (< ENTREES1 > ;
*< SORTIES1 >, *, < SORTIES1 >) |*
< IDENTIFICATEUR DE SOUS-UNITE > < DUPLICATION >
*(< ENTREES1 > ; < SORTIES1 >, *, < SORTIES1 >)*

AFFECTATIONS SIMPLES

< CHARGEMENT DE REGISTRE > ::=
< VARIABLE DE TYPE REGISTRE > <= < EXPRESSION BOOLEENNE > |
< VARIABLE DE TYPE REGISTRE > < = ! (< EXPRESSION ARITHMETIQUE >) |
< VARIABLE DE TYPE ETAT > <= < VARIABLE DE TYPE ETAT > |
< VARIABLE DE TYPE ETAT > <= < ETIQUETTE >
< AFFECTATION D'HORLOGE > ::=
< VARIABLE DE TYPE HORLOGE > := < EXPRESSION BOOLEENNE >
< AFFECTATION D'ENTIER > ::=
< VARIABLE DE TYPE ENTIER > $= < EXPRESSION ARITHMETIQUE > |
< VARIABLE DE TYPE TABLEAU ENTIER > $= < EXPRESSION ARITHMETIQUE > |
plusun (< VARIABLE DE TYPE COMPTEUR >)|
moinsun (< VARIABLE DE TYPE COMPTEUR >)
< APPEL DE PROCEDURE > ::= appel < IDENTIFICATEUR DE PROCEDURE FORTRAN > |
appel < IDENTIFICATEUR DE PROCEDURE FORTRAN >
(< LISTE DE PARAMETRES FORTRAN >) |
appel < IDENTIFICATEUR DE PROCEDURE ASSEMBLEUR >
appel < IDENTIFICATEUR DE PROCEDURE ASSEMBLEUR >
(< LISTE DE PARAMETRES ASSEMBLEUR >)
< LISTE DE PARAMETRES FORTRAN > ::= < PARAMETRE FORTRAN > |
< PARAMETRE FORTRAN >, < LISTE DE PARAMETRES FORTRAN >
< PARAMETRE FORTRAN > ::= < CONSTANTE ARITHMETIQUE > |
< VARIABLE DE TYPE COMPTEUR > |
< VARIABLE DE TYPE ENTIER > |
< VARIABLE DE TYPE TABLEAU ENTIER >

< LISTE DE PARAMETRES ASSEMBLEUR > ::= < PARAMETRE ASSEMBLEUR > |
< PARAMETRE ASSEMBLEUR >, < LISTE DE PARAMETRE ASSEMBLEUR >

< PARAMETRE ASSEMBLEUR > ::= < PARAMETRE FORTRAN > |
< VARIABLE DE TYPE REGISTRE > |
< VARIABLE DE TYPE SIGNAL >

< *INSTRUCTION ALLERA* > *::=* *allera* < *EXPRESSION D'ETAT* > |
allera < *EXPRESSION D'ETAT* > *de* < *IDENTIFICATEUR DE SOUS-UNITE* > |
allera < *EXPRESSION D'ETAT* > *de* < *IDENTIFICATION DE SOUS-UNITE* >< *DUPLICATION* >

INSTRUCTION FAIRE

< *INSTRUCTION FAIRE* > *::=* *faire* < *EXPRESSION D'ETAT* >

INSTRUCTION ET AFFECTATIONS CONDITIONNELLES

α → INSTRUCTION / AFFECTATION

< α *CONDITIONNELLE SIMPLE* > *::=*
si < *EXPRESSION SI* > *alors* < *ALORS* α > *sinon* < *SINON* α > |
si < *EXPRESSION SI* > *alors* < *ALORS* α > |
sie < EXPRESSION LOGIQUE > alors < ALORS α > sinon < SINON α > |
sie < EXPRESSION LOGIQUE > alors < ALORS α >

< *ALORS* α > *::=* < α *SIMPLE* > | (< *LISTE D'*α *CONDITIONNABLES* >)

< *LISTE D'AFFECTATIONS CONDITIONNABLES* > *::=* < *AFFECTATION CONDITIONNABLE* > |
< *LISTE D'AFFECTATIONS CONDITIONNABLES* >, < *AFFECTATION CONDITIONNABLE* >

< *LISTE D'INSTRUCTIONS CONDITIONNABLES* > *::=* < *INSTRUCTION CONDITIONNABLE* > |
< *LISTE D'INSTRUCTIONS CONDITIONNABLES* > ; < *INSTRUCTION CONDITIONNABLE* >

< *SINON* α > *::=* < α *CONDITIONNABLE* > | (< *LISTE D'*α *CONDITIONNABLES* >)

< *AFFECTATION CONDITIONNABLE* > *::=* < *AFFECTATION CONDITIONNELLE* > |
< *AFFECTATION SIMPLE* >

< *INSTRUCTION CONDITIONNABLE* > *::=* < *INSTRUCTION FAIRE* > |
< *INSTRUCTION CONDITIONNELLE* > |
< *BRANCHEMENT SIMPLE* > |
< *INSTRUCTION D'AFFECTATION* >

< α *CONDITIONNELLE VECTORIELLE* > *::=*
si < *EXPRESSION SI* > *alors* < *VECTEUR* α > *sinon* < *VECTEUR* α > |
si < *EXPRESSION SI* > *alors* < *VECTEUR* α > |
sie < EXPRESSION LOGIQUE > alors < VECTEUR α > sinon < VECTEUR α > |
sie < EXPRESSION LOGIQUE > alors < VECTEUR α >

< *VECTEUR* α > *::=* () | < *VECTEUR* α > () | (< *LISTE D'*α *CONDITIONNABLES* >) |
< *VECTEUR* α > (< *LISTE D'*α *CONDITIONNABLES* >)

LES ITERATIONS

< *ITERATION* > *::=* < *EN-TETE D'ITERATION* > *début*
< *LISTE DE BRANCHEMENTS* > *fin*

< *EN-TETE D'ITERATION* > *::=* < *POUR* > < *INDEX D'ITERATION* > < *DEPUIS* >
< *VALEUR INITIALE* > *pas* < *INCREMENT* > *a* < *LIMITE* > |
< *POUR* > < *INDEX D'ITERATION* > < *DEPUIS* > < *VALEUR INITIALE* > *a* < *LIMITE* >

< *POUR* > *::=* *pour* | **pourtout**

< *DEPUIS* > *::=* *depuis* | = | *de*

< *INDEX D'ITERATION* > *::=* < *IDENTIFICATEUR* >

< *VALEUR INITIALE* > *::=* < *EXPRESSION D'ITERATION* >

< *INCREMENT* > *::=* < *EXPRESSION D'ITERATION* >

< *LIMITE* > *::=* < *EXPRESSION D'ITERATION* >

< *INSTRUCTION CONDITIONNELLE D'ITERATION* > *::=*
sia < *EXPRESSION CONDITIONNELLE D'ITERATION* > *alors*
début < *LISTE DE BRANCHEMENTS* > *fin* |
sia < *EXPRESSION CONDITIONNELLE D'ITERATION* > *alors*
début < *LISTE DE BRANCHEMENTS* > *fin sinon*
début < *LISTE DE BRANCHEMENTS* > *fin*

< *AFFECTATION ITERATIVE* > *::=* < *EN-TETE D'ITERATION* > *début*
< *LISTE D'AFFECTATIONS* > *fin*

< *AFFECTATION CONDITIONNELLE D'ITERATION* > *::=*
sia < *EXPRESSION CONDITIONNELLE D'ITERATION* > *alors*
début < *LISTE D'AFFECTATIONS* > *fin* |
sia < *EXPRESSION CONDITIONNELLE D'ITERATION* > *alors*
début < *LISTE D'AFFECTATIONS* > *fin sinon début*
< *LISTE D'AFFECTATIONS* > *fin*

LES EXPRESSIONS

Une expression est soit une expression simple, soit une expression conditionnelle, soit une expression logique.

< *EXPRESSION* > *::=* < **EXPRESSION LOCIQUE** > |
< *EXPRESSION LOGIQUE D'ETAT* > |
< *EXPRESSION BOOLEENNE* > |
< **EXPRESSION ARITHMETIQUE** > |
< *EXPRESSION D'ETAT* > |
< *EXPRESSION D'ITERATION* > |
< *EXPRESSION LOGIQUE D'ITERATION* > |
< *EXPRESSION D'HORLOGE* >

< *EXPRESSION* β > *::=* < *EXPRESSION CONDITIONNELLE* β > |
< *EXPRESSION* β *SIMPLE* >

β → *BOOLEENNE* | ARITHMETIQUE | *D'ETAT* | *D'ITERATION*.

EXPRESSION BOOLEENNE SIMPLE

< *EXPRESSION BOOLEENNE SIMPLE* > *::=* < *TERME BOOLEEN* > |
< *TERME BOOLEEN* > + < *EXPRESSION BOOLEENNE SIMPLE* >

< *TERME BOOLEEN* > *::=* < *FACTEUR BOOLEEN* > |
< *FACTEUR BOOLEEN* > . < *TERME BOOLEEN* >

< *FACTEUR BOOLEEN* > *::=* < *SECONDAIRE BOOLEEN* > |
< *SECONDAIRE BOOLEEN* > < *OPREL* > < *FACTEUR BOOLEEN* >

< *SECONDAIRE BOOLEEN* > *::=* < *PRIMAIRE BOOLEEN* > |
< *PRIMAIRE BOOLEEN* > ε < *SECONDAIRE BOOLEEN* >

< *PRIMAIRE BOOLEEN* > *::=* < *OP MONADIQUE BOOLEEN* > < *PRIMAIRE BOOLEEN* > |
< *OP MONADIQUE DE TRANSFORMATION* > < *PRIMAIRE BOOLEEN* > |
(< *EXPRESSION BOOLEENNE SIMPLE* >) |
< *CONSTANTE BOOLEENNE* > |
< *VARIABLE DE TYPE SIGNAL* > |
< *VARIABLE DE TYPE REGISTRE* > |

< *OPREL* > *::=* < | ≤ | > | ≥ | = | ≠

EXPRESSION ARITHMETIQUE SIMPLE ET D'ITERATION SIMPLE

γ → ARITHMETIQUE | *D'ITERATION*.

< *EXPRESSION* γ *SIMPLE* > *::=* < *TERME* γ > |
< *TERME* γ > + < *EXPRESSION* γ *SIMPLE* > |
< *TERME* γ > − < *EXPRESSION* γ *SIMPLE* >

< *TERME* γ > *::=* *FACTEUR* γ > |
< *FACTEUR* γ > . < *TERME* γ > |
< *FACTEUR* γ > ./ < *TERME* γ > |
< *FACTEUR* γ > *rem* < *TERME* γ >

< *FACTEUR D'ITERATION* > *::=* *abs* < *FACTEUR D'ITERATION* > |
− < *FACTEUR D'ITERATION* > |
(< *EXPRESSION D'ITERATION* >) |
< *ENTIER SANS SIGNE* > |
< *INDEX D'ITERATION* >

< FACTEUR ARITHMETIQUE > ::= abs < FACTEUR ARITHMETIQUE > |
- < FACTEUR ARITHMETIQUE > |
(< EXPRESSION ARITHMETIQUE SIMPLE >) |
< CONSTANTE ARITHMETIQUE > |
< VARIABLE DE TYPE ENTIER > |
< VARIABLE DE TYPE TABLEAU ENTIER > |
< VARIABLE DE TYPE COMPTEUR > |
$ (< EXPRESSION BOOLEENNE SIMPLE >) |
< INDEX D'ITERATION >

EXPRESSION D'HORLOGE

< EXPRESSION D'HORLOGE > ::= < EXPRESSION D'HORLOGE SIMPLE > |
si < EXPRESSION SI > alors < EXPRESSION D'HORLOGE SIMPLE >
sinon < EXPRESSION D'HORLOGE >

< EXPRESSION D'HORLOGE SIMPLE > ::= < TERME HORLOGE > |
< TERME HORLOGE > + < EXPRESSION D'HORLOGE SIMPLE >

< TERME HORLOGE > ::= < FACTEUR HORLOGE > |
< FACTEUR HORLOGE > ε < TERME HORLOGE >

< FACTEUR HORLOGE > ::= < VARIABLE DE TYPE HORLOGE >
(< EXPRESSION D'HORLOGE >) |
** R ↑ < CONSTANTE ARITHMETIQUE > ↑ < FACTEUR D'HORLOGE >*
** D ↑ < CONSTANTE ARITHMETIQUE > ↑ < FACTEUR D'HORLOGE >*
/ + < FACTEUR HORLOGE >

EXPRESSIONS LOGIQUE D'ETAT ET LOGIQUE D'ITERATION

σ → D'ETAT | D'ITERATION

< EXPRESSION LOGIQUE σ > ::= < TERME LOGIQUE σ > |
< TERME LOGIQUE σ > / < EXPRESSION LOGIQUE σ >

< TERME LOGIQUE σ > ::= < FACTEUR LOGIQUE σ > |
< FACTEUR LOGIQUE σ > ε < TERME LOGIQUE σ >

< FACTEUR LOGIQUE D'ETAT > ::= ↑ < EXPRESSION LOGIQUE D'ETAT > ↑ |
< EXPRESSION D'ETAT SIMPLE > = < EXPRESSION D'ETAT SIMPLE > |
< EXPRESSION D'ETAT SIMPLE > ≠ < EXPRESSION D'ETAT SIMPLE >

*< EXPRESSION D'ETAT SIMPLE > ::= < ETIQUETTE > | * |*
< ETIQUETTE EXTERNE > | < VARIABLE DE TYPE D'ETAT > |
état de < IDENTIFICATEUR DE SOUS-UNITE > |
état de < IDENTIFICATEUR DE SOUS-UNITE > < DUPLICATION >
< FACTEUR LOGIQUE D'ITERATION > ::= ↑ < EXPRESSION LOGIQUE D'ITERATION > ↑ |
< EXPRESSION D'ITERATION SIMPLE >
< OPREL >
< EXPRESSION D'ITERATION SIMPLE >

EXPRESSION LOGIQUE

< EXPRESSION LOGIQUE > ::= < TERME LOGIQUE > |
< TERME LOGIQUE > ou < EXPRESSION LOGIQUE >
< TERME LOGIQUE > ::= < FACTEUR LOGIQUE > |
< FACTEUR LOGIQUE > et < TERME LOGIQUE >
< FACTEUR LOGIQUE > ::= ↑ < EXPRESSION LOGIQUE > ↑ |
< EXPRESSION ARITHMETIQUE SIMPLE > < OPREL >
< EXPRESSION ARITHMETIQUE SIMPLE >

EXPRESSIONS CONDITIONNELLES

μ → *BOOLEENNE* | ARITHMETIQUE | *D'ETAT*

< EXPRESSION CONDITIONNELLE μ > ::=
si < EXPRESSION SI > alors < EXPRESSION μ SIMPLE >
sinon < EXPRESSION μ > |
sie < EXPRESSION LOGIQUE > alors < EXPRESSION μ SIMPLE >
sinon < EXPRESSION μ >
< EXPRESSION SI > ::= < EXPRESSION BOOLEENNE SIMPLE > |
↑ < EXPRESSION LOGIQUE D'ETAT > ↑
< EXPRESSION CONDITIONNELLE D'ITERATION > ::= sia
< EXPRESSION LOGIQUE D'ITERATION > alors
< EXPRESSION D'ITERATION SIMPLE > sinon
< EXPRESSION D'ITERATION >

LES CONSTANTES -

CONSTANTES BOOLEENNES

On ne fait pas de distinction entre le symbole 0 et la valeur logique faux, ni entre le symbole 1 et la valeur logique vrai.

< *CONSTANTE BOOLEENNE* > *::=* < *CONSTANTE SCALAIRE* > |
< *CONSTANTE VECTEUR* > |
< *CONSTANTE TABLEAU* >

< *CONSTANTE SCALAIRE* > *::= 0 | 1*

< *CONSTANTE VECTEUR* > *::=* < *CONSTANTE SCALAIRE* > |
< *CONSTANTE VECTEUR* > < *CONSTANTE SCALAIRE* >

< *CONSTANTE TABLEAU* > *::=* < *CONSTANTE VECTEUR* > |
< *CONSTANTE TABLEAU* >, < *CONSTANTE VECTEUR* >

CONSTANTES ARITHMETIQUES

Les constantes arithmétiques sont les entiers positifs et négatifs.
Leur valeur absolue ne doit pas dépasser $2^{31}-1$.

< CONSTANTE ARITHMETIQUE > ::= < ENTIER SANS SIGNE > |
< SIGNE > < ENTIER SANS SIGNE >

< *ENTIER SANS SIGNE* > *::=* < *CHIFFRE* > |
< *ENTIER SANS SIGNE* > < *CHIFFRE* >

< *SIGNE* > *::= + | -*

ETIQUETTES

< *ETIQUETTE* > *::=* < *IDENTIFICATEUR* >

LES ELEMENTS TERMINAUX

On peut répartir les symboles de base du langage de la façon suivante :

< *SYMBOLE DE BASE* > ::= < *LETTRE* > | < *CHIFFRE* > | < *DELIMITEUR* >

< *DELIMITEUR* > ::= < *OPERATEUR* > | < *SEPARATEUR* > |
< *DECLARATEUR* > | < *SPECIFIEUR* >

< *OPERATEUR* > ::= < *OPERATEUR MONADIQUE* > | < *OPERATEUR DYADIQUE* >

< *OPERATEUR MONADIQUE* > ::= < *OP MONADIQUE ARITHMETIQUE* > |
< *OP MONADIQUE BOOLEEN* > |
< OP MONADIQUE DE CONVERSION > |
< *OP MONADIQUE DE TRANSFORMATION* >

< *OPERATEUR DYADIQUE* > ::= < *OP DYADIQUE ARITHMETIQUE* > |
< *OP DYADIQUE BOOLEEN* > | < OP DYADIQUE LOGIQUE > |
< *CONCATENATION* >

< *CONCATENATION* > ::= ε

< *OP DYADIQUE BOOLEEN* > ::= + | . | < *OPREL* >

< *OPREL* > ::= < | ≤ | = | > | ≥ | ≠

< *OP DYADIQUE ARITHMETIQUE* > ::= + | - | . | ./ | *rem*

< OP DYADIQUE LOGIQUE > ::= et | ou

< *OP MONADIQUE DE TRANSFORMATION* > ::= *P | *R | *D

< OP MONADIQUE DE CONVERSION > ::= $ | !

< *OP MONADIQUE BOOLEEN* > ::= / < *OP DYADIQUE BOOLEEN* > | -

< *OP MONADIQUE ARITHMETIQUE* > ::= - | abs | plusun | moinsun

< *SPECIFIEUR* > ::= *horlogemère*

< *DECLARATEUR* > ::= *unité* | *externe* | *signal* | *registre* |
état | *horloge* | compteur | entier |
tableau | procédure | fortran

< *SEPARATEUR* > ::= , | ; | : | (|) | ↑ | " | := | <= | %= |
début | *fin* | *pour* | *depuis* | *pas* | *a* | *faire* |
allera | *si* | sie | *sia* | appel | pourtout | *de*

< *IDENTIFICATEUR* > ::= < *LETTRE* > |
< *IDENTIFICATEUR* > < *LETTRE* > |
< *IDENTIFICATEUR* > < *CHIFFRE* >

< *LETTRE* > ::= *A* | *B* | ..... *Y* | *Z*

< *CHIFFRE* > ::= *0* | *1* | ..... *8* | *9*

Des commentaires peuvent être insérés à n'importe quel endroit d'un programme LASCAR. Il doivent être enfermés entre des guillemets (symbole : ").

Tous les mots clé ont une traduction anglaise. De plus, les mots clé plusun et moinsun peuvent être abrégés respectivement en p1 et m1.

LISTE DES MOTS CLE

Français	Anglais
A	TO
ABS	ABS
ALLERA	GOTO
APPEL	CALL
COMPTEUR	COUNTER
DEBUT	BEGIN
DEPUIS	FROM
ENTIER	INTEGER
ETAT	STATE
EXTERNE	EXTERNAL
FAIRE	DO
FIN	END
FORTRAN	FORTRAN
HORLOGE	CLOCK
HORLOGEMERE	MASTERCLOCK
MOINSUN - M1	MINUSONE - M1
PAS	STEP
PLUSUN - P1	PLUSONE - P1
POUR	FOR
POURTOUT	FORALL
PROCEDURE	PROCEDURE
REGISTRE	REGISTER
REM	REM
SI	IF
SIA	ANIF
SIE	INIF
SIGNAL	SIGNAL
TABLEAU	ARRAY
UNITE	UNIT

ANNEXE II

SIGNATURES DES INSTRUCTIONS DU CODE IBM/360

On notera :

- EXEC1, EXEC2, EXE3 : les temps d'exécution comptés en nombre de micro-instructions.
- * : une demande de lecture
- Δ : une demande d'écriture

Les temps d'attente, dus à des conflits éventuels pour l'accès au bus, ou à la nécessité d'un échange avec la mémoire principale, n'apparaissent pas dans les signatures. De plus, une signature caractérisant uniquement l'interprétation d'une instruction présente dans le processeur, l'algorithme d'appel de l'instruction, qui dépend de la longueur de l'instruction précédemment exécutée, ne fait pas partie de la signature.

Les instructions sont regroupées en 10 types, en fonction de la structure de leur signature :

TYPE	SIGNIFICATION
0	EXEC1 suivi d'un branchement
1	EXEC1
2	EXEC1 * EXEC2
3	EXEC1 Δ EXEC2
4	EXEC1 * EXEC2 * EXEC3
5	EXEC1 * EXEC2 Δ EXEC3
6	test and set (TS)
7	store multiple (STM)
8	load multiple (LM)
9	appel superviseur (SVC)
10	invalidation partielle en fin de processus (ce n'est pas une instruction du code 360)

INSTRUCTIONS RR (1/2 mot)

CODE	INSTRUCTION	SIGNATURE	TYPE
04	SPM	6	1
05	BALR	2 ou 4	0 ou 1
06	BCTR	1 ou 3	0 ou 1
07	BCR	2 ou 3	0 ou 1
08	SSK	3 Δ 1	3
09	ISK	3 * 4	2
0A	SVC	4 Δ 2 Δ 2 * 2 * 1	9
10	LPR	3	1
11	LNR	3	1
12	LTR	3	1
13	LCR	3	1
14	NR	3	1
15	CLR	3	1
16	OR	3	1
17	XR	3	1
18	LR	2	1
19	CR	3	1
1A	AR	3	1
1B	SR	3	1
1C	MR	160	1
1D	DR	180	1
1E	ALR	3	1
1F	SLR	3	1

INSTRUCTIONS RX (1 mot)

CODE	INSTRUCTION	SIGNATURE	TYPE
40	STH	oper Δ 1	3
41	LA	oper	1
42	STC	oper Δ 1	3
43	IC	oper * 3	2
44	EX	oper * 3	2
45	BAL	1 ou oper + 2	0 ou 1
46	BCT	1 ou oper + 2	0 ou 1
47	BC	1 ou oper + 2	0 ou 1
48	LH	oper * 1	2
49	CH	oper * 4	2
4A	AH	oper * 4	2
4B	SH	oper * 4	2
4C	MH	oper * 64	2
50	ST	oper Δ 1	3
54	N	oper * 4	2
55	CL	oper * 4	2
56	O	oper * 4	2
57	X	oper * 4	2
58	L	oper * 1	2
59	C	oper * 4	2
5A	A	oper * 4	2
5B	S	oper * 4	2
5C	M	oper * 160	2
5D	D	oper * 180	2
5E	AL	oper * 4	2
5F	SL	oper * 4	2
		oper : calcul opérande 3 ou 4 micro-instructions suivant que le registre index est nul ou non.	

INSTRUCTIONS RS ET SI (1 mot)

CODE	INSTRUCTION	SIGNATURE	TYPE
80	SSM	2 * 4	2
82	LPSW	2 * 2 * 3	4
84	WRD	2 * 4	2
85	RDD	5 Δ 1	3
86	BXH		0 ou 1
	R1 pair	7 ou 8	
	R1 impair	6 ou 7	
87	BXLE		0 ou 1
	R1 pair	7 ou 8	
	R1 impair	6 ou 7	
88	SRL	6 + 2 n	1
89	SLL	6 + 2 n	1
8A	SRA	7 + 2 n (R1 > 0)	1
8B	SLA	8 + 4 n + 6	1
8C	SRDL	13 + 4 n + 5	1
8D	SLDL	13 + 4 n + 5	1
90	STM	mx(4 Δ) 2	7
91	TM	2 * 4,5	2
92	MVI	3 Δ 1	3
93	TS	3 TS 1 Δ 1	6
94	NI	2 * 2 Δ 1	5
95	CLI	2 * 4,5	2
96	OI	2 * 2 Δ 1	5
97	XI	2 * 2 Δ 1	5
98	LM	mx(4 *) 2	8
		n : 0 à 4, deux derniers bits de l'adresse	
		m = R3 - R1 + 1 si R3 ≥ R1	
		m = R3 - R1 + 17 si R3 < R1	

ANNEXE III

MODELE DU PROCESSEUR

```
. S.    TEXTE SOURCE

0001    'UNITE' MICROP (H,HETAT,OK;STOP,LEC,ECR,TS,INV,RAZ);
0045    'HORLOGEMERE' H,HETAT;
0054    'COMPTEUR' NUMPROC, "NUMERO DU PROCESSEUR"
0063               SING,SINGL,SINGE,   "POINTS SINGULIERS"
0080            K,      "COMPTEUR PARTIEL D ATTENTE"
0082                DELAI,  "COMPTEUR DE CYCLES"
0088             INTPS,     "TEMPS SEPARANT 2 APPELS MEMOIRE CONSECUTIFS"
0094                  CICLES,       "COMPTEUR DE CYCLES"
0101                NBINS,     "COMPTEUR NOMBRE D'INSTRUCTIONS"
0107                KREPOS,     "COMPTEUR CYCLES INACTIVITE"
0114               KLECI,   "COMPTEUR LECTURES INSTRUCTIONS"
0120             ALECI,  "ATTENTE LECTURE INSTRUCTION"
0126            KLECOP,  "COMPTEUR LECTURE OPERANDE"
0133            ALECOP,   "ATTENTE LECTURE OPERANDE"
0140            KECROP,   "COMPTEUR ECRITURE OPERANDE"
0147            AECROP,   "ATTENTE ECRITURE OPERANDE"
0154            KTES,      "COMPTEUR TEST AND SET"
0159            ATES,      "ATTENTE SUR TEST AND SET DEMANDE"
0164            KINV,       "COMPTEUR INVALIDATION PARTIELLE"
0169            AINV;       "ATTENTE SUR INVALIDATION PARTIELLE"
0174    'ETAT' ETX(0:12);   "ADRESSES BRANCHEMENT EN FONCTION DES CODE OP."
0185    'ENTIER' ADINS, "ADRESSE INSTRUCTION"
0192             TYPE,  "TYPE INSTRUCTION"
0197             EXEC1,EXEC2,EXEC3 ,  "TEMPS EXEC DE L'INSTRUCTION"
0215             LIMITE, "CADRAGE DE L'INSTRUCTION"
0222             LONG1, LONG;   "LONGUEUR DE L'INSTRUCTION"
0233    'TABLEAU' 'ENTIER' HISTOGR(1:64),     "HISTOGRAMME DES TEMPS
0249                                  SEPARANT 2 APPELS MEMOIRE CONSECUTIFS"
0249                    RALEC(0:63),     "HISTOGRAMME DES ATTENTES EN LECTURE"
0261                    RAECR(0:63);     "HISTOGRAMME DES ATTENTES EN ECRITURE"
0273    'TABLEAU' 'ENTIER' INSTYP(0:10);      "NOMBRE D'APPELS DES DIFFERENTS
0288                                        TYPES D'INSTRUCTIONS"
0288    'PROCEDURE' NEXTI(7);      "VA CHERCHER LA PROCHAINE INSTRUCTION"
0298    'FORTRAN' PINSTY(1);      "IMPRIME INSTYP"
0309    'FORTRAN' STAT(14),ETATP(13),SIMPC(2),HISTO(2),HISTOL(2),HISTOE(2);
0367       "SOUS-PROGRAMMES D'IMPRESSION DE STATISTIQUES"
0367    <HETAT> 'SI' -STOP 'ALORS' ('CALL' ETATP (NUMPROC,NBINS,KREPOS,KLECI,
0416         ALECI,KLECOP,ALECOP,KECROP,AECROP,KTES,ATES,KINV,AINV));
0472    <H>  'P1' (CICLES);
0485    LECTI1: <H>   'P1'(KLECI),'P1' (INTPS),  'ALLERA' LECTI11;
0522    LECTI11: LEC:=1; 'FAIRE' HISTOIRE; <H> 'ALLERA' LECTI12;
0558    LECTI12: 'SI' OK 'ALORS' (LEC:=0; 'FAIRE' HISLEC);
0586             <H> 'SI' OK 'ALORS' ('ALLERA' TESTNIA1, 'P1'(INTPS))
0613           'SINON' ('P1'(ALECI), 'P1'(K));
0630    TESTNIA1: <H> 'SIE' LIMITE = 0  'ALORS' DELAI >=1
0659                          'SINON' DELAI $=4,
0668                                  'P1' (INTPS),
0677                    'ALLERA' ATTENTE;
0686    ATTENTE: <H> 'M1'(DELAI),
0706                                  'P1' (INTPS),
```

```
00715                'SIE' DELAI = 0 'ALORS' 'ALLERA' CODEOP;
00732   CODEOP: <H>   'ALLERA' ETX(TYPE), 'P1' (INTPS),
00762              INSTYP(TYPE) $=INSTYP(TYPE)+1;
00790   TYPE0: <H> 'M1'(EXEC1),'P1'(INTPS), 'SIE' EXEC1=1 'ALORS'
00826           'ALLERA' PREP0;
00833   PREP0: <H> 'P1'(NBINS),'CALL' SIMPC(CICLES,NUMPROC),
00874       'CALL' NEXTI(NUMPROC,TYPE,ADINS,LONG,EXEC1,EXEC2,EXEC3),
00924         LIMITE $= ADINS 'REM' 4,
00939                                  'P1' (INTPS),
00948         'SIE' LIMITE=0 'ALORS' (DELAI $= 3, 'ALLERA' VERSA)
00974              'SINON' ( 'P1'(KLECI), 'ALLERA' LECTI0 );
00994   LECTI0: LEC:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' LECTI02;
01029   LECTI02: 'SI' OK  'ALORS' (LEC:=0; 'FAIRE' HISLEC);
01057            <H> 'SI' OK 'ALORS' (DELAI $=3, 'P1'(INTPS), 'ALLERA' VERS
01089            'SINON' ('P1'(ALECI), 'P1'(K));
01106   TYPE1: <H> 'M1'(EXEC1),
01124                                  'P1' (INTPS),
01133              'SIE' EXEC1 = 0 'ALORS' 'ALLERA' PREP1;
01149   PREP1: <H> LONG1 $= LONG, 'P1'(NBINS),
01178               'CALL' SIMPC(CICLES,NUMPROC),   "IMPRESSION D'UNE TRACE
01201        'CALL' NEXTI(NUMPROC,TYPE,ADINS,LONG,EXEC1,EXEC2,EXEC3),
01251                      "LECTURE PROCHAINE INSTRUCTION SUR UN FICHIER"
01251             LIMITE $= ADINS 'REM' 4,     "CADRAGE INSTRUCTION"
01266                  'P1' (INTPS),
01275         'SIE' LIMITE=0 'ALORS' (DELAI $= 2, 'ALLERA' VERSA)
01301         'SINON' ('SIE' LONG1=1 'ALORS' (DELAI $= 3, 'ALLERA' VERSD)
01328                  'SINON' (DELAI $=5, 'ALLERA' VERSD));
01347   VERSA: <H> 'M1'(DELAI),'P1'(INTPS), 'SIE' DELAI=0 'ALORS' 'ALLERA'
01384              LECTI1;
01391   VERSD: <H> 'M1'(DELAI),'P1'(INTPS), 'SIE' DELAI=0 'ALORS'
01427                  ('SIE' LONG=1 'ALORS' 'ALLERA' VERSB
01442                                  'SINON' 'ALLERA' LECTI1);
01452   VERSB: <H> 'P1'(INTPS), 'ALLERA' CODEOP;
01478   TYPE2: <H> 'P1'(INTPS), 'M1'(EXEC1),
01505               'SIE' EXEC1=0 'ALORS'
01514                        ('P1'(KLECOP),  'ALLERA' TYPE2L);
01534   TYPE2L: LEC:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST2L;
01568   TEST2L: 'SI' OK 'ALORS'
01579                     (LEC:=0; 'FAIRE' HISLEC;
01594                     <H> 'P1'(INTPS),  'SIE' EXEC2=0 'ALORS' 'ALLERA'
01621                        'SINON' 'ALLERA' SUITEX2)
01631               'SINON' (<H>  'P1'(ALECOP), 'P1'(K));
01652   SUITEX2: <H>  'M1'(EXEC2),
01672                                  'P1' (INTPS),
01681                  'SIE' EXEC2=0 'ALORS' 'ALLERA' PREP1;
01697   TYPE3: <H> 'M1'(EXEC1),
01715                                  'P1' (INTPS),
01724              'SIE' EXEC1=0 'ALORS'
01733                   ('P1'(KECROP),  'ALLERA' TYPE3E);
01753   TYPE3E: ECR:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST3E;
01787   TEST3E: 'SI' OK 'ALORS' (ECR:=0; 'FAIRE' HISECR; <H>  'ALLERA' PREP
01823            'P1'(INTPS))
01832             'SINON' (<H>  'P1'(AECROP), 'P1'(K));
01853   TYPE4: <H> 'M1'(EXEC1),
01871                                  'P1' (INTPS),
01880            'SIE' EXEC1=0 'ALORS'
01889             ('P1'(KLECOP), 'ALLERA' TYPE4L);
01909   TYPE4L: LEC:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST4L;
```

```
01943   TEST4L: <H> 'SI' OK 'ALORS' ('ALLERA' TYPE4X2, 'P1'(INTPS))
01976              'SINON' ('P1'(ALECOP), 'P1'(K));
01994           'SI' OK 'ALORS' (LEC:=0; 'FAIRE' HISLEC);
02014   TYPE4X2: <H> 'M1'(EXEC2),
02034                                  'P1' (INTPS),
02043               'SIE' EXEC2=0 'ALORS'
02052               ( 'P1'(KLECOP), 'ALLERA' TYPE4L2);
02073   TYPE4L2: LEC:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST4L2;
02109   TEST4L2: 'SI' OK 'ALORS' (LEC:=0; 'FAIRE' HISLEC;
02136             <H> 'P1'(INTPS),  'SIE' EXEC3=0 'ALORS' 'ALLERA' PREP1
02163                              'SINON' 'ALLERA' SUITEX3)
02173             'SINON' (<H>  'P1'(ALECOP), 'P1'(K));
02194   SUITEX3: <H> 'M1'(EXEC3),
02214                                  'P1' (INTPS),
02223                 'SIE' EXEC3=0 'ALORS' 'ALLERA' PREP1;
02239   TYPE5: <H> 'M1'(EXEC1),
02257                                  'P1' (INTPS),
02266             'SIE' EXEC1=0 'ALORS'
02275            ('P1'(KLECOP),  'ALLERA' TYPE5L);
02295   TYPE5L: LEC:=1;'FAIRE'HISTOIRE; <H> 'ALLERA' TEST5L;
02329   TEST5L:   <H> 'SI' OK 'ALORS' ('ALLERA' TYPE5X, 'P1'(INTPS))
02361        'SINON' ('P1'(ALECOP), 'P1'(K));
02379           'SI' OK 'ALORS' (LEC:=0; 'FAIRE' HISLEC);
02399   TYPE5X: <H> 'M1'(EXEC2),
02418                                  'P1' (INTPS),
02427             'SIE' EXEC2=0 'ALORS' ('P1'(KECROP), 'ALLERA' TYPE5E );
02456   TYPE5E: ECR:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST5E;
02490   TEST5E: 'SI' OK 'ALORS' (ECR:=0; 'FAIRE' HISECR; <H>  'ALLERA' PREP1,
02526             'P1'(INTPS))
02535          'SINON' (<H>  'P1'(AECROP), 'P1'(K));
02556   TYPE6: <H> 'M1'(EXEC1),
02574                                  'P1' (INTPS),
02583             'SIE' EXEC1=0 'ALORS'
02592              ('P1'(KTES), 'ALLERA' TYPE6T);
02610   TYPE6T: TS:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST6;
02642   TEST6: 'SI' OK 'ALORS' (TS:=0; <H> 'P1'(INTPS), 'P1'(KECROP),
02680                      'ALLERA' TYPE5E)
02688        'SINON' (<H>  'P1'(ATES));
02702   TYPE7: <H> 'M1'(EXEC1),
02720                                  'P1' (INTPS),
02729        'SIE' EXEC1=0 'ALORS' ('P1'(KECROP), 'ALLERA' TYPE7E);
02758   TYPE7E: ECR:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST7E;
02792   TEST7E:<H> 'SI' OK 'ALORS' ('ALLERA' TYPE7SUI, 'P1'(INTPS))
02826             'SINON' ('P1'(AECROP), 'P1'(K));
02844           'SI' OK 'ALORS' (ECR:=0; 'FAIRE' HISECR);
02864   TYPE7SUI: <H> EXEC1 $= 5, 'M1'(EXEC3),
02893                                  'P1' (INTPS),
02902             'SIE' EXEC3_<0 'ALORS' ('M1'(EXEC2), 'ALLERA' SUITEX2)
02930              'SINON' 'ALLERA' TYPE7;
02938   TYPE8: <H> 'M1'(EXEC1),
02956                                  'P1' (INTPS),
02965            'SIE' EXEC1=0 'ALORS' ('P1'(KLECOP), 'ALLERA' TYPE8L);
02994   TYPE8L: LEC:=1;'FAIRE' HISTOIRE;<H> 'ALLERA' TEST8L;
03028   TEST8L:<H> 'SI' OK 'ALORS' ('ALLERA' TYPE8SUI, 'P1'(INTPS))
03062        'SINON' ('P1'(ALECOP), 'P1'(K));
03080           'SI' OK 'ALORS' (LEC:=0; 'FAIRE' HISLEC);
```

```
03100   TYPE8SUI: <H> EXEC1 $= 5, 'M1'(EXEC3),
03129                                 'P1' (INTPS),
03138             'SIE' EXEC3=0 'ALORS' ('M1'(EXEC2), 'ALLERA' SUITEX2)
03166             'SINON' 'ALLERA' TYPE8;
03174   TYPE9: <H> 'M1'(EXEC1),
03192                                 'P1' (INTPS),
03201              'SIE' EXEC1=0 'ALORS' ('P1'(KECROP), 'ALLERA' TYPE9E);
03230   TYPE9E: ECR:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST9E;
03264   TEST9E:<H> 'SI' OK 'ALORS' ('ALLERA' TYPE9X, 'P1'(INTPS))
03296       'SINON' ('P1'(AECROP), 'P1'(K));
03314           'SI' OK 'ALORS' (ECR:=0; 'FAIRE' HISECR);
03334   TYPE9X: <H> 'M1'(EXEC2),
03353                                 'P1' (INTPS),
03362              'SIE' EXEC2=0 'ALORS' ('P1'(KECROP), 'ALLERA' TYPE9E2);
03392   TYPE9E2: ECR:=1;'FAIRE' HISTOIRE; <H> 'ALLERA' TEST9E2;
03428   TEST9E2:<H> 'SI' OK 'ALORS' ('ALLERA' TYPE9X2, 'P1'(INTPS))
03462           'SINON' ('P1'(AECROP), 'P1'(K));
03480         'SI' OK 'ALORS' (ECR:=0; 'FAIRE' HISECR);
03500   TYPE9X2: <H> EXEC1 $= 1, EXEC2 $= 2,
03527                                 'P1' (INTPS),
03536              'ALLERA' TYPE4;
03543   TYPE10: INV:=1;'FAIRE' HISTOIRE;
03566         <H> 'P1'(KINV), 'ALLERA' TYPE10SUI;
03588   TYPE10SUI: <H> 'SI' OK 'ALORS' 'ALLERA' TYPE10FIN
03615                  'SINON' 'P1'(AINV);
03624             'SI' OK 'ALORS' INV:=0;
03634   TYPE10FIN:<H> 'ALLERA' REPOS;
03654   HISTOIRE:<H> 'SIE' INTPS > 64 'ALORS'
03676              'P1'(SING)
03683                                 'SINON'
03684              HISTOGR(INTPS)$=HISTOGR(INTPS)+1, INTPS$=0;
03724   HISLEC: <H>  'SIE' K > 63 'ALORS'
03740            'P1'(SINGL) 'SINON'
03749         RALEC(K)$=RALEC(K)+1, K$=0;
03773   HISECR: <H>     'SIE' K >  63 'ALORS'
03789            'P1'(SINGE)  'SINON'
03798        RAECR(K)$=RAECR(K)+1 , K$=0;
03822   REPOS: LEC:=0;
03834          ECR:=0;
03840          TS:=0;
03845          INV:=0;
03851          RAZ:=0;
03857     <H> 'CALL' STAT(NUMPROC,NBINS,KREPOS,KLECI,ALECI,KLECOP,
03906          ALECOP,KECROP,AECROP,KTES,ATES,KINV,AINV,EXEC1),
03954   'CALL' SIMPC(CICLES,NUMPROC),
03977   'CALL' HISTO(HISTOGR,SING),
03998   'CALL' HISTOL(RALEC,SINGL),
04019   'CALL' HISTOE(RAECR,SINGE),
04040   'CALL' PINSTY(INSTYP),
04056   'ALLERA' STOPH;
04063   STOPH: STOP:=1;
04063   STOPH: STOP:=1;
04076   <H>  'ALLERA' HALT;
04085   HALT: ;
```

MODELE DU CACHE - PREMIERE VERSION : UNE FILE D'ATTENTE

```
U. S.   TEXTE SOURCE

00001   'UNITE' FCACHE (H,HETAT,DEM(0:4,1:12),OKMEM(1:3);ECHMEM(1:3),OK(1:12));
00064   'HORLOGEMERE' H,HETAT;
00073   'TABLEAU' 'ENTIER'
00075                LFILE(0:12),     "LONGUEUR FILE D'ATTENTE"
00087                FDM(1:12),       "FILE DES DEMANDES MEMOIRE. ON Y MET LE NO
00097                                     DES PROCESSEURS"
00097              FDTYP(1:12);     "FILE DES TYPES DE DEMANDES MEMOIRE"
00109   'REGISTRE' INHIBE(1:12),      "1-> LA DEMANDE DU PROC I PEUT ETRE PRISE"
00123              TBLOK(1:12,1:12),      "REPONSES POSITIVES A 1 PROC"
00140              ACCESM;      "1: ACCES MEMOIRE EN COURS"
00147   'ENTIER' NBP,        " NB DE PROCESSEURS"
00152            LMEM,     "NB DE LIGNES -1 DE MEM CACHE"
00157            AIG,      "VARIABLE DE CONTROLE AIGUILLAGE"
00161            BOOL,     "REPONSE PROCEDURE SELECTE1"
00166            DEMAND,      "RECUPERE LES DEMANDEURS, POUR 1 ACTION VERS CACHE"
00173            REPMEM,      "ENREGISTREMENT D' 1 REPONSE MEMOIRE"
00180            LBLOC, TRAV,      "VARIABLES DE TRAVAIL"
00191            TPSLEC,      "TEMPS -2 DE LECTURE D'1 BLOC"
00198            RAND,ICIBLOC;      "POUR GENERATION NB ALEATOIRE"
00211   'COMPTEUR' CIRCUL,      "DERNIERE DEMANDE PRISE EN COMPTE"
00219              INV,          "COMPTEUR DES LIGNES INVALIDEES"
00223            C1,C2,
00229              INACTIF,       "COMPTEUR CYCLES D'INACTIVITE"
00237            PTFDM;      "POINTEUR FILE DES DEMANDES MEMOIRE"
00243   'ETAT' ECLATM(0:9),  "NOREP,REPLEC,REPECR,REPTES"
00256                      "CALLMEM,LIRE,ECRIRE,TES,INVAL,RAZ"
00256          ETINV(3:4);    "FININV,ASKTES"
00267   'FORTRAN' DECALE(3),ETATC(9),MOYEN(1);
00296   'PROCEDURE' SELECTE(3),SELECTE1(4),RANDOM(2);
00330   <HETAT>  'CALL' ETATC (FDM,FDTYP,DEMAND,REPMEM,ICIBLOC,
00376                    CIRCUL,INV,INACTIF,PTFDM);
00402   <H>  LFILE(PTFDM)$=LFILE(PTFDM)+1;
00433    'SI' OKMEM 'ALORS'
00440                (ECHMEM:=000; <H> REPMEM $= 1)
00464                (ECHMEM:=000; <H> REPMEM $= 2)
00488                (ECHMEM:=000; <H> REPMEM $= 3);
```

```
00513   SELECT: OK:=000000000000;
00536       <H> DEMAND $= $(INHIBE.(/+DEM)),
00564           'SIE' DEMAND = 0 'ALORS'
00574               ( TRAV $= FDM(1),
00587                 'SI' ACCESM 'ALORS' AIG $= REPMEM
00605                 'SINON'
00606                  ('SIE' PTFDM>0 'ALORS' AIG $= 4
00621                   'SINON' (AIG $= 0, 'P1'(INACTIF))))
00642           'SINON'
00643                    ('CALL' SELECTE (CIRCUL,NBP,DEMAND),
00672                     'SI' DEM(,CIRCUL) 'ALORS'
00686                         (AIG $= 5)
00693                         (AIG $= 6)
00700                         (AIG $= 7)
00707                         (AIG $= 8)
00714                         (AIG $= 9, INV $= LMEM));
00732          'FAIRE' ECLATM(AIG);
00745   NOREP:<H> 'ALLERA' SELECT;
00762   REPLEC: <H> 'ALLERA' LIRBLOC,  LBLOC $= TPSLEC ;
00794   LIRBLOC:  'SIE' LBLOC=0 'ALORS'
00811               (OK(TRAV):=1;
00823                <H> INHIBE(TRAV) <=1, ACCESM <=0,
00850                    REPMEM $= 0,
00859                    'CALL' DECALE (FDM,FDTYP,PTFDM),
00884                    'ALLERA' SELECT)
00892               'SINON' (<H> 'M1'(LBLOC));
00907   REPECR: OK(TRAV):=1;
00925           <H> INHIBE(TRAV)<=1, ACCESM<=0,
00952               REPMEM $= 0,
00961               'CALL' DECALE (FDM,FDTYP,PTFDM),
00986              AIG $= 0,
00992               'ALLERA' SELECT;
01000   REPTES:  OK(TRAV):=1;
01018            <H> INHIBE(TRAV) <=1, REPMEM $= 0, AIG $= 0,
01051                'CALL' DECALE (FDM,FDTYP,PTFDM),
01076                'ALLERA' ATTENTP;
01085   ATTENTP:  'SI' DEM(1,TRAV) 'ALORS'
01106              (ECHMEM(2):=1; <H> 'ALLERA' ATTENTM);
01132              OK(TRAV):=0;
01143   ATTENTM: 'SI' OKMEM(2) 'ALORS'
01161              (OK(TRAV) :=1;
01173               <H> REPMEM $=0, INV $=LMEM,
01194                   ACCESM <=0, 'ALLERA' SELECT);
01212   LIRE: <H> 'CALL' RANDOM (RAND,ICIBLOC),
01242             'ALLERA' FINLIRE,
01251          'SIE' ICIBLOC = 0 'ALORS'
01262             ( 'P1'(PTFDM),
01272               FDTYP(PTFDM) $=1, FDM(PTFDM) $= CIRCUL,
01305               INHIBE(CIRCUL) <=0);
01323   FINLIRE: 'SIE' ICIBLOC = 1 'ALORS' OK:=TBLOK(,CIRCUL);
01360            'FAIRE' SELECT1;
```

```
01369   SELECT1: <H> 'SI' ACCESM 'ALORS' AIG $= REPMEM
01398                'SINON'
01399               ('SIE' PTFDM>0 'ALORS' AIG $= 4
01414                'SINON'
01415                 (DEMAND $= $(INHIBE.(/+DEM)),
01441                  'SIE' DEMAND = 0 'ALORS' AIG $= 0
01456                  'SINON'
01457                    ('CALL' SELECTE1(CIRCUL,NBP,DEMAND,BOOL),
01492                     'SIE' BOOL=0 'ALORS' AIG $= 0  'SINON'
01506                      ('SI' DEM(,CIRCUL) 'ALORS'
01521                          (AIG $= 5)
01528                          (AIG $= 6)
01535                          (AIG $= 7)
                               (AIG $= 8)
                               (AIG $= 9, INV $= LMEM )))))),
                TRAV $= FDM(1);
                'FAIRE' ECLATM(AIG);
           E:<H> 'P1'(PTFDM),
                 FDM(PTFDM) $= CIRCUL, FDTYP(PTFDM) $=2,
                 INHIBE(CIRCUL) <=0, 'ALLERA' SELECT1;
           <H> 'P1'(PTFDM),
               FDM(PTFDM) $= CIRCUL, FDTYP(PTFDM) $= 3,
               INHIBE(CIRCUL) <=0, 'ALLERA' SELECT1;
            :<H> 'P1'(PTFDM),
                 FDM(PTFDM) $= CIRCUL,
                 FDTYP(PTFDM) $= 4,
01799            INHIBE(CIRCUL) <= 0,
01816             'ALLERA' SELECT1;
01825   RAZ: 'SIE' INV = -1 'ALORS'
01837              (OK(CIRCUL):=1;
01851               <H> INV $= LMEM, 'ALLERA' SELECT)
01871          'SINON' (<H> 'M1'(INV));
01884   CALLMEM: 'SIE' FDTYP(1) < 3 'ALORS'
01904              (ECHMEM(FDTYP(1)):=1;
01924               <H> ACCESM<=1, 'ALLERA' SELECT)
01944             'SINON'
01945              ('SIE' INV> -1 'ALORS'
01954                     (<H> 'M1'(INV), 'ALLERA' SELECT)
01973                'SINON' 'FAIRE' ETINV(FDTYP(1)));
01992   FININV: OK(TRAV):=1;
02010            <H> INHIBE(TRAV)<=1, ACCESM<=0,
02037                'CALL' DECALE(FDM,FDTYP,PTFDM),
02062                INV $= LMEM, 'ALLERA' SELECT;
02079   ASKTES: ECHMEM(3):=1;
02098           <H> ACCESM<=1, 'ALLERA' SELECT;
```

MODELE DU CACHE - DEUXIEME VERSION : DEUX FILES D'ATTENTE

U. S. TEXTE SOURCE

```
00001 'UNITE' PCACHE (H,HETAT,DEM(0:4,1:12),OKMEM(1:3);ECHMEM(1:3),OK(1:12));
00064 'HORLOGEMERE' H,HETAT;
00073 'TABLEAU' 'ENTIER'
00075              LFILE(0:12),     "LONGUEUR FILE D'ATTENTE"
00087           LFILE1(0:12),  "LONG. FILE DE TES"
00100           TPC(0:119),    "INTERTPS APPEL PMEM PAR CACHE"
00111              FDM(1:12),       "FILE DES DEMANDES MEMOIRE. ON Y MET LE NO
00121       DES PROCESSEURS"
00121            FDM1(1:12),   "FILE DEMAND DE TES"
00132             FDTYP(1:12) ;     "FILE DES TYPES DE DEMANDES MEMOIRE"
00144 'REGISTRE' INHIBE(1:12),      "1-> LA DEMANDE DU PROC I PEUT ETRE PRISE"
00158              TBLOK(1:12,1:12),      "REPONSES POSITIVES A 1 PROC"
00175              ACCESM;      "1: ACCES MEMOIRE EN COURS"
00182 'ENTIER' NBP,        " NB DE PROCESSEURS"
00187           LMEM,     "NB DE LIGNES -1 DE MEM CACHE"
00192           AIG,      "VARIABLE DE CONTROLE AIGUILLAGE"
00196           BOOL,     "REPONSE PROCEDURE SELECTE1"
00201           DEMAND,      "RECUPERE LES DEMANDEURS, POUR 1 ACTION VERS CACHE
00208           REPMEM,      "ENREGISTREMENT D' 1 REPONSE MEMOIRE"
00215           LBLOC, TRAV,      "VARIABLES DE TRAVAIL"
00226           TPSLEC,      "TEMPS -2 DE LECTURE D'1 BLOC"
00233           RAND,ICIBLOC;      "POUR GENERATION NB ALEATOIRE"
00246 'COMPTEUR' CIRCUL,      "DERNIERE DEMANDE PRISE EN COMPTE"
00254              INV,          "COMPTEUR DES LIGNES INVALIDEES"
00258           C1,C2,C3,ACC1,    "DEMANDE INVAL EN COURS"
00272         SING,OUTPC,    "POINTS SINGULIERS"
00283       PTFDM1,     "PTEUR FILE DES TES"
00290              INACTIF,       "COMPTEUR CYCLES D'INACTIVITE"
00298           PTFDM;      "POINTEUR FILE DES DEMANDES MEMOIRE"
00304 'ETAT' ECLATM(0:10) ;  "NOREP,REPLEC,REPECR,REPTES"
00318                      "CALLMEM,LIRE,ECRIRE,TES,INVAL,RAZ,FINV"
00318 'FORTRAN' DECALE(3),ETATC(9),MOYEN(1),DECAL1(2),HISTOC(3);
00367 'PROCEDURE' SELECTE(3),SELECTE1(4),RANDOM(2);
00401 <HETAT>  'CALL' ETATC (FDM,FDTYP,DEMAND,REPMEM,ICIBLOC,
00447                        CIRCUL,INV,INACTIF,PTFDM),
00473          'CALL' MOYEN(LFILE1),'CALL' HISTOC(TPC,SING,OUTPC),
00512      'CALL' MOYEN(LFILE) ;
00526 <H>  LFILE(PTFDM) $=LFILE(PTFDM)+1,
00557    LFILE1(PTFDM1) $=LFILE1(PTFDM1)+1,
00589     'SI' (/=ECHMEM(1:3))=0 'ALORS' 'P1'(C3);
00614  'SI' OKMEM 'ALORS'
00621               (ECHMEM(1:3):=000; <H> REPMEM .= 1)
00650               (ECHMEM(1:3):=000; <H> REPMEM .= 2)
00679               (ECHMEM(1:3):=000; <H> REPMEM .= 3);
```

```
00709   SELECT: 'SIE' AIG#2 'ALORS' OK:=000000000000;
00739       <H> DEMAND $= $(INHIBE(1:12).(/+DEM(0:4,))),
00779           'SIE' DEMAND = 0 'ALORS'
00789   ('SI' ACCESM 'ALORS'
00798       (AIG$='SIE' REPMEM=0 'ALORS' 10.ACC1
00820                'SINON' REPMEM,
00828           TRAV$='SIE' AIG=10 'ALORS' FDM1(1)
00848                'SINON' FDM(1))
00856    'SINON'
00857      ('SIE' PTFDM  > 0 'ALORS' (AIG$=4,TRAV$=FDM(1))
00886     'SINON'
00887   ('SIE' ACC1=0 'ALORS'
00896       (AIG$=0,'P1'(INACTIF),INHIBE(1:8)<=11111111)
00935         'SINON' (TRAV$=FDM1(1),AIG$=10)))
00960           'SINON'
00961                   ('CALL' SELECTE (CIRCUL,NBP,DEMAND),
00990                    'SI' DEM(,CIRCUL) 'ALORS'
01004                        (AIG $= 5)
01011                        (AIG $= 6)
01018                        (AIG $= 7)
01025                        (AIG $= 8)
01032                        (AIG $= 9 ));
01041           'FAIRE' ECLATM(AIG);
01054   NOREP: <H> 'ALLERA' SELECT;
01071   REPLEC: <H> 'ALLERA' LIRBLOC,  LBLOC $= TPSLEC ;
01103   LIRBLOC:  'SIE' LBLOC=0 'ALORS'
01120               (OK(TRAV):=1;
01132                <H> INHIBE(TRAV) <=1, ACCESM <=0,
01159                    REPMEM $= 0,
01168                    'CALL' DECALE (FDM,FDTYP,PTFDM),
01193                    'ALLERA' SELECT)
01201             'SINON' (<H> 'M1'(LBLOC));
01216   REPLCR: OK(TRAV):=1;
01234           <H> INHIBE(TRAV)<=1, ACCESM<=0,
01261               REPMEM $= 0,
01270               'CALL' DECALE (FDM,FDTYP,PTFDM),
01295               'ALLERA' SELECT;
01303   REPTES:     <H> INHIBE(TRAV) <=1, REPMEM $= 0, AIG $= 0,
01343               'CALL' DECALE (FDM,FDTYP,PTFDM),
01368               'ALLERA' OKTES;
01375   OKTES: OK(TRAV):=1;
01392         <H>   'ALLERA' ATTENTP;
01404   ATTENTP:  'SI' DEM(1,TRAV) 'ALORS'
01425             (ECHMEM(2):=1; <H> 'ALLERA' ATTENTM,
01450        TPC(C3) $=TPC(C3)+1,C3$=0);
01474              OK(TRAV):=0;
01485   ATTENTM: <H> 'SI' OKMEM(2) 'ALORS'
01506   ('P1'(PTFDM1),FDM1(PTFDM1)$=TRAV,INHIBE(TRAV)<=0,
01550         ACC1$=1,REPMEM$=0,
01566                  ACCESM <=0, 'ALLERA' SELECT);
01584   LIRE: <H> 'CALL' RANDOM (RAND,ICIBLOC),
01614             'ALLERA' FINLIRE, AIG $= 0 ,
01629           'SIE' ICIBLOC = 0 'ALORS'
01640              ( 'P1'(PTFDM),
01650                FDTYP(PTFDM) $=1, FDM(PTFDM) $= CIRCUL,
01683                INHIBE(CIRCUL) <=0);
01701   FINLIRE: 'SIE' ICIBLOC = 1 'ALORS' OK:=TBLOK(,CIRCUL);
01738            'FAIRE' SELECT1;
```

```
01747   SELECT1: <H>     AIG$=0,
01764    'SI' ACCESM 'ALORS' (AIG $= REPMEM ,TRAV$=FDM(1))
01796                 'SINON'
01797                ('SIE' PTFDM>0 'ALORS' (AIG $= 4,TRAV$=FDM(1))
01826                 'SINON'
01827      ('SIE' AIG=0 'ALORS'
01835          ('SIE' ACC1=1 'ALORS'
01844              (AIG$=10,TRAV$=FDM1(1)))
01866       'SINON'
01867                  (DEMAND $= $(INHIBE.(/+DEM)),
01893                   'SIE' DEMAND = 0 'ALORS' AIG $= 0
01908                   'SINON'
01909                     ('CALL' SELECTE1(CIRCUL,NBP,DEMAND,BOOL),
01944                      'SIE' BOOL=0 'ALORS' AIG $= 0  'SINON'
01958                       ('SI' DEM(,CIRCUL) 'ALORS'
01973                            (AIG $= 5)
01980                            (AIG $= 6)
01987                            (AIG $= 7)
01994                            (AIG $= 8)
02001                            (AIG $= 9 ))))));
02014             'FAIRE' ECLATM(AIG);
02027   ECRIRE:<H> 'P1'(PTFDM),
02046              FDM(PTFDM) $= CIRCUL, FDTYP(PTFDM) $=2,
02079              INHIBE(CIRCUL) <=0, 'ALLERA' SELECT1;
02105   TES:<H> 'P1'(PTFDM),
02121           FDM(PTFDM) $= CIRCUL, FDTYP(PTFDM) $= 3,
02154           INHIBE(CIRCUL) <=0, 'ALLERA' SELECT1;
02180   INVAL:<H> 'P1'(PTFDM1),
02199             FDM1(PTFDM1) $= CIRCUL,
02219        ACC1$=1,
02226             INHIBE(CIRCUL) <= 0,
02243              'ALLERA' SELECT1;
02252   RAZ: 'SIE' INV = -1 'ALORS'
02264              (OK(CIRCUL):=1;
02278               <H> INV $= LMEM, 'ALLERA' SELECT)
02298           'SINON' (<H> 'M1'(INV));
02311   CALLMEM: DCHMEM(FDTYP(1)):=1;
02338              <H> ACCESM<=1, 'ALLERA' SELECT,
02358    'SIE' C3<120 'ALORS'
02366        TPC(C3) $=TPC(C3)+1 'SINON'
02384         ('P1'(SING),OUTPC$=OUTPC+C3),C3$=0;
02414   FINV:<H>  'SIE' INV  > 0 'ALORS'
02429             ('M1'(INV),'ALLERA' SELECT)
02445             'SINON' 'ALLERA' FININV;
02454   FININV: OK(TRAV):=1;
02472             <H> INHIBE(TRAV)<=1,
02490       'CALL' DECAL1(FDM1,PTFDM1),
02511                INV $= LMEM, 'ALLERA' SELECT,
02528     'SIE' PTFDM1=0 'ALORS' ACC1$=0;
02545
```

SIMULATION DU MODELE

```
load lascar modele
R; T=1.76/2.49 18:27:05

start initial modele
EXECUTION BEGINS...
.* ceci est un commentaire
.* appel d'une macro commande d'initialisation
.mm
*INITIALISATION DU MODELE....
R H=1
UNITE PCACHE
*  APPEL D'UNE MACRO COMMANDE D'INITIALISATION DU CACHE
INITPCA
* ON SUPPRIME L'IMPRESSION DE CETTE COMMANDE
IMP OFF
UNITE MODELE
UNITE MICROP 1
IND NUMPROC=1
*APPEL D'UNE MACRO COMMANDE D'INITIALISATION DU PROCESSEUR
INITMICR
IMP OFF
UNITE MODELE
UNITE MICROP 2
IND NUMPROC=2
* INITIALISATION DU 2EME PROCESSEUR
INITMICR
IMP OFF
UNITE MODELE
.* on demande un cycle. une procedure de trace imprime les cycles
.* auxquels un des processeurs lit une nouvelle instruction
.c
CYCLES=         1PROCESS=     2
CYCLES=         1PROCESS=     1
.* on se met dans l'environnement du processeur 1 et on imprime
.* les parametres renvoyes par nexti
.unite microp 1
.outd type, long, exec1, exec2, exec3
 2
-
 2
-
 4
-
 7
-
0
.outh adins
00025F88
.outd limite, delai
0
-
 2
```

```
.c 3
.p microp
LECTI11
.* le processeur est dans l'etat lecti11. il demande une lecture
.p lec
1
.c
.* dans l'etat suivant, on teste l'acquittement venant du cache
.p ok
0
.c
.p ok
0
.* la donnee n'est pas dans le cache. il faut attendre le temps d'un
.* echange avec la memoire principale
.c 7
 . p ok
0
.p microp
LECTI12
.c
.p ok
1
.* la donnee est arrivee. c'est une lecture instruction
.* on la charge et on la decode
.c 3
.p microp
CODEOP
.*etat de decodage sur type instruction
.c
.p microp
TYPE2
.* on decompte exec1
.c 4
.p microp
TYPE2L
.* demande de lecture. a l'etat suivant on teste le signal OK
.c
.p microp, ok
TEST2L
-
1
.* cette fois la donnee etait dans le cache: reponse immediate
.* on decompte exec2
.c 7
 . c
?.c
CYCLES=       32PROCESS=    1
.* le processeur 1 a lu l'instruction suivante
.* on force la fin de simulation du processeur 1, comme si il etait
.* arrive en fin de trace
.r microp=type5fin
.c
.p microp
REPOS
```

```
.c
ARRET PROCESSEUR**   1 **
NBINS=        2  KREPOS=        0  KLECI=        1
ALECI=        9  KLECOP=        1  ALECOP=        0
KECROP=       0  AECROP=        0  KTES=         0
ATES=         0  KINV=          0  AINV=          0
NONEXEC=      2
CYCLES=       34PROCESS=     1
HISTOGRAMME INTER-TEMPS APPELS MEMOIRE

      0       0       0       1       0       0       0       1
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
       0

REPARTITION ATTENTE LECTURE SUR CACHE
      1       0       0       0       0       0       0       0
      0       1       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
       0
REPARTITION ATTENTE ECRITURE SUR CACHE

      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
      0       0       0       0       0       0       0       0
       0
REPARTITION TYPES INSTRUCTIONS
    0     0     1     0     0     0     0     0     0     0     0
.* ces resultats ont ete sortis par les procedures FORTRAN de l'etat REPOS
.* on arrete la simulation
.fin
R; T=2.40/4.06 18:56:55
```

ANNEXE IV

ROUTINES DE L'INTERPRETEUR DE SIMULATION

Les littéraux qui apparaissent dans les formules donnant le nombre exact d'instructions déroulées lors de l'appel de certaines routines ont la signification suivante :

p : nombre de paramètres de la routine
no : numéro de duplication d'une unité
nbex : nombre d'exemplaires d'une unité

Pour une variable quelconque de type tableau

a_1, a_2, a_3, ... : étendue des dimensions
d : nombre de dimensions + 2

Pour une variable booléenne de type tableau

n = nombre de bits / 8 si la division est sans reste
nombre de bits / 8 + 1 si la division a un reste
m = a_1 / 8 si la division est sans reste
m = a_1 / 8 + 1 si la division a un reste

Pour une variable entière de type tableau

ne : nombre d'éléments du tableau

NUMERO	NOM	SIGNIFICATION	NOMBRE EXACT D'INSTRUCTIONS EXECUTEES	NBRE TYPIQUE D'INSTRUCTIONS
0	NOP	Instruction vide	1	1
1	BRANCHR	Saut dans la chaîne si tour en R	4 ou 6	5
2	BRANCHS	Saut dans la chaîne si tour en S	4 ou 6	5
3	BRANCHC	Saut dans la chaîne conditionnel	6 ou 8	7
4	BRANCH	Saut dans la chaîne inconditionnel	4	4
5	BRANCHP	Saut dans la chaîne à la valeur en sommet de pile 1	4	4
6	ENTREEU	Début d'unité	16	16
7	SORTIEU	Fin d'unité	Pas de duplication : 12, avec duplication 31	16
8	TERMINE	Fin de chaîne	Tour en R : 5, Tour en S : 8 si stabilisation 18 si pas encore stable	12
9	DUPLICX	Initialisation d'une unité englobée	32 + 4 no	35
10	ADDITION	Addition de 2 entiers sur pile 1	5	5
11	SUBSTRACT	Soustraction de 2 entiers sur pile 1	5	5
12	MULTIPLI	Multiplication de 2 entiers sur pile 1	5	5
13	DIVISION	Division de 2 entiers sur pile 1	7	7
14	RESTDIV	Reste de la division de 2 entiers sur pile 1	7	7
15	CHANGSI	Sommet pile 1 ← - sommet pile 1	4	4
16	AD1	Sommet pile 1 ← sommet pile 1 + 1	4	4
17	PERMUT	Permutation des 2 éléments en sommet de pile 1	6	6
18	DOUBL1	Ré-empile le sommet de pile 1	3	3
19	DOUBL2	Ré-empile les 2 demi-mots au sommet de pile 1	4	4
20	DECAPN	Dépile 1/2 mot de pile 1	2	2
21	VALNUM	Calcul sur pile 1 de $	-	-

NUMERO	NOM	SIGNIFICATION	NOMBRE EXACT D'INSTRUCTIONS EXECUTEES	NBRE TYPIQUE D'INSTRUCTIONS
22	SELECT1	Sélection d'un bit d'un être booléen	30	30
23	DECAPT	Dépile un être de pile 1	3	3
24	MASK	Génère un être booléen dont tous les bits sont à 1, sur pile 1	-	-
25	EMPVAL	Empile une valeur immédiate sur pile 1	4	4
26	EMPENTI	Empile un index de "boucle pour" sur pile 1	7	7
27	EMPUTAT	Empile une variable d'état de l'unité sur pile 1, mode total	8	8
28	EDEMIMU	Empile une variable d'état de l'unité sur pile 1, mode partiel	25 + 10 d	35
29	EMPILIU	Empile un signal de l'unité sur pile 1, mode total	21 + 5 (n + d)	46
30	EMPIP	Empile un signal de l'unité sur pile 1, mode partiel		285
31	EMPILM1U	Empile un registre de l'unité sur pile 1, mode total	22 + 5 (n + d)	47
32	EMPMP	Empile un registre de l'unité sur pile 1, mode partiel		254
33	RANGENTI	Range un index de "boucle pour" dans TABLEI, à partir de pile 1	7	7
34	RANGUTA	Range une variable d'état de l'unité à partir de pile 1, dans TABLEM2, mode total	13 ou 17	17
35	RDEMIMU	Range une variable d'état de l'unité à partir de pile 1, dans TABLEM2, mode partiel	35 + 10 d	45
36	RANGIUT	Range un signal de l'unité à partir de pile 1, dans TABLEI, mode total	22 + 26 n	74
37	RGTIP	Range un signal de l'unité à partir de pile 1, dans TABLEI, mode partiel		337
38	RANGMUT	Range un registre de l'unité à partir de pile 1, dans TABLEM2, mode total	18 + 37 n	73

NUMERO	NOM	SIGNIFICATION	NOMBRE EXACT D'INSTRUCTIONS EXECUTEES	NBRE TYPIQUE D'INSTRUCTIONS
39	RGTMP	Range un registre de l'unité à partir de pile 1, dans TABLEM2, mode partiel		418
40	RGHYT	Range une horloge de l'unité à partir de pile 1, dans TABLEI, mode total	74 + 34 (n - 1)	74
41	RGHYP	Range une horloge de l'unité à partir de pile 1, dans TABLEI, mode partiel	-	-
42	EMPXTAT	Empile un état d'une unité différente sur pile 1	8	8
43	RANGXTA	Range un état d'une unité différente à partir de pile 1	12 ou 15	15
44	EMPILIX	Empile un signal d'une unité différente sur pile 1, mode total	22 + 5 (n + d)	42
45	RANGIX	Range un signal d'une unité différente à partir de pile 1, mode total	21 + 27 n	75
46	EMPILM1X	Empile un registre d'une unité différente sur pile 1, mode total	22 + 5 (n + d)	-
47	RANGMXT	Range un registre d'une unité différente à partir de pile 1, mode total	17 + 38 n	-
48	CONCATEN	Concaténation	-	-
49	RGTXHY	Range une horloge d'une unité différente à partir de pile 1, mode total	74 + 34 (n - 1)	74
50	DECALAGE	Décalage d'un être booléen	-	-
51	ROTATION	Décalage circulaire d'un être booléen	-	-
52	NEGTENS	Calcul de l'opposé d'un être booléen	6 + 5 n	11
53	ENTREUHZ	Initialise le traitement des remises à zéro d'horloges internes	12	12
54	TRANSP12	Transposition des 2 premières dimensions d'un être booléen	-	-

NUMERO	NOM	SIGNIFICATION	NOMBRE EXACT D'INSTRUCTIONS EXECUTEES	NBRE TYPIQUE D INSTRUCTIONS
55	TRANSPIJ	Transposition de 2 dimensions quelconques d'un être booléen	-	-
56	REDUCOU	Opérateur /+ sur des êtres booléens		628
57	REDUCET	Opérateur /. sur des êtres booléens		136
58	REDUCSUP	Opérateur /> sur des êtres booléens		-
59	REDUCINF	Opérateur /< sur des êtres booléens		-
60	REDUCSUE	Opérateur /≥ sur des êtres booléens		-
61	REDUCINE	Opérateur /≤ sur des êtres booléens		-
62	REDUCEG	Opérateur /= sur des êtres booléens		-
63	REDUCDIF	Opérateur /≠ sur des êtres booléens		-
64	SUP	Opérateur > sur des êtres booléens	19 + 8 n	-
65	INF	Opérateur < sur des êtres booléens	19 + 8 n	-
66	SUPEG	Opérateur ≥ sur des êtres booléens	19 + 8 n	-
67	INFEG	Opérateur ≤ sur des êtres booléens	19 + 8 n	-
68	EGAL	Opérateur = sur des êtres booléens	19 + 8 n	27
69	DIFF	Opérateur ≠ sur des êtres booléens	19 + 8 n	-
70	OU	Opérateur + sur des êtres booléens	19 + 8 n	-
71	ET	Opérateur . sur des êtres booléens	19 + 8 n	35
72	PLUGRAN	Opérateur > sur des index de boucle	13 ou 14	14
73	PLUPETIT	Opérateur < sur des index de boucle	13 ou 14	14
74	PLUGREG	Opérateur ≥ sur des index de boucle	13 ou 14	14
75	PLUEGAL	Opérateur ≤ sur des index de boucle	13 ou 14	14
76	EGALMO	Opérateur = sur des index de boucle	13 ou 14	14
77	DIFFMO	Opérateur ≠ sur des index de boucle	13 ou 14	14

NUMERO	NOM	SIGNIFICATION	NOMBRE EXACT D'INSTRUCTIONS EXECUTEES	NBRE TYPIQUE D INSTRUCTION
78	EMPBX	Empile base d'une unité externe sur pile 1	8	8
79	DECAPBX	Range base d'une unité externe depuis pile 1	8	8
80	DUPLICU	Génère les bases d'une unité donnée et de numéro de duplication donné	32 + 4 no	35
81	BASEU	Génère les bases d'une unité donnée	21	21
82	SORTSP	Retour au superviseur	3	3
83	RETZ	Remet à zéro une horloge interne	$2(p-1) + 10p + 3 + \sum_{1}^{p} 4\, n_i$	33
84	PERMRDR	Enlève bits de présence et transfère de table M2 à table M1 un registre	$5 + 17p + \sum_{1}^{p} 8\, n_i$	224
85	PERMRDE	Idem pour une variable d'état	$5 + 17p + \sum_{1}^{p} 6\, ne_i$	96
86	PRESY	Enlève bits de présence d'un signal ou horloge	$2(p-1) + 16p + 1 + \sum_{1}^{p} 10\, n_i$	185
87	SCRIPT	Impression d'une variable booléenne	-	-
88	PROGMOD	Propagation d'un indicateur de modification	pas de modification : 3, modification : 7	6
89	SORTUHZ	Fin de traitement des remises à zéro d'horloges internes	10 ou 19	12
90	ENZX	Initialise base externe avant appel de PRESYX	17	17
91	PRESYX	Enlève bits de présence d'un signal ou horloge d'unité externe	$10p + \sum_{1}^{p} (13 + 10\, n_i)\, nbex_i$	581
92	DJUMP	Ré-empile l'avant dernier élément de pile 1	5	5
93	ABS	Calcule la valeur absolue du sommet de pile 1	4	4

NUMERO	NOM	SIGNIFICATION	NOMBRE EXACT D'INSTRUCTIONS EXECUTEES	NBRE TYPIQUE D INSTRUCTIONS
94	AEMPV	Empile un index de "boucle pour" sur pile 2	13	13
95	CRANG	Range un entier à partir de pile 2	14	14
96	CEMPV	Empile un entier sur pile 2	12	12
97	CEMP	Empile une constante sur pile 2	9	9
98	CADD	Addition de 2 tableaux entiers ou de 2 entiers	scalaire : 10, tableau 10 + 6 ne	10
99	CMOINS	Soustraction de 2 tableaux entiers ou de 2 entiers	scalaire : 10, tableau 10 + 6 ne	-
100	CMULT	Multiplication de 2 tableaux entiers ou de 2 entiers	scalaire : 12, tableau 13 + 9 ne	12
101	CDIV	Division entière de 2 tableaux entiers ou de 2 entiers	scalaire : 15, tableau 10 + 11 ne	-
102	CREST	Reste de la division entière de 2 tableaux entiers ou de 2 entiers	scalaire : 15, tableau 10 + 11 ne	15
103	CNEG	Calcul de l'opposé d'un tableau entier ou d'un entier	3 + 5 ne	-
104	CINF	Opérateur < sur des entiers	18	18
105	CSUP	Opérateur > sur des entiers	18	18
106	CINFG	Opérateur ≤ sur des entiers	18	18
107	CSUPG	Opérateur ≥ sur des entiers	18	18
108	CDIFF	Opérateur ≠ sur des entiers	18	18
109	CEGAL	Opérateur = sur des entiers	18	18
110	PLUS1	Incrémentation d'un compteur	12	12
111	MOINS1	Décrémentation d'un compteur	12	12
112	EINF	Opérateur < sur des tableaux entiers	14 + 7 ne	21
113	ESUP	Opérateur > sur des tableaux entiers	14 + 7 ne	21

NUMERO	NOM	SIGNIFICATION	NOMBRE EXACT D'INSTRUCTIONS EXECUTEES	NBRE TYPIQUE D INSTRUCTION
114	EINFG	Opérateur ≤ sur des tableaux entiers	14 + 7 ne	21
115	ESUPG	Opérateur ≥ sur des tableaux entiers	14 + 7 ne	21
116	EDIFF	Opérateur ≠ sur des tableaux entiers	14 + 7 ne	21
117	EEGAL	Opérateur = sur des tableaux entiers	14 + 7 ne	21
118	CEMPVA	Empile un entier sur pile 1	8	8
119	VERIF	Vérifie que le sommet de pile 1 est compris entre les bornes déclarées d'une dimension	10	10
120	CVALNUM	Opérateur $ général	$33+6d+2m+(37+18m+8\ \min(a_1,32))\ a_2\ a_2 \ldots$	212
121	VALBIN	Opérateur ! sur des tableaux entiers mode total	$91+13d+ne(28+3\ \max(m-4,0)+8\ \min(d_1,32)+29\ \min(m,4))$	
122	VALBINP	Opérateur ! sur des tableaux entiers mode partiel	$96+17d+ne(28+3\ \max(m-4,0)+8\ \min(d_1,32)+29\ \min(m,4))$	
123	CABS	Valeur absolue d'un tableau entier	3 + 5 ne	-
124	EMPILET	Empile un tableau entier sur pile 2, mode total	26 + 4 ne	-
125	RANGET	Range un tableau entier depuis pile 2, mode total	23 + 4 ne	-
126	EMPILEP	Empile un tableau entier sur pile 2, mode partiel	$62 + 18d + 4\ ne + (3 + 3\ d_2)\ d_3$	84
127	RANGEP	Range un tableau entier depuis pile 2, mode partiel	$60 + 18d + 4\ ne + (3 + 3\ d_2)\ d_3$	82
128	EEMPVAP	Empile un élément de tableau entier sur pile 1	26 + 29 (d - 1)	26

NUMERO	NOM	SIGNIFICATION	NOMBRE EXACT D'INSTRUCTIONS EXECUTEES	NBRE TYPIQUE D INSTRUCTIONS
129	EEMPVAT	Empile un entier scalaire, déclaré tableau, sur pile 1	9	9
130	BRANCE	Saut dans la chaîne conditionné par un élément de comparaison entre 2 vecteurs entiers	9 ou 11	10
131	BRANC	Saut dans la chaîne conditionné par une comparaison entre 2 entiers scalaires	6 ou 8	7
132	NEGAT	Remplace sur pile 1 n booléens par leur opposé	7 + 6 ne	13
133	CALLP	Appel de sous-programme assembleur	22 si 0 paramètre, 27 + 12p sinon	87
134	CALLF	Appel de sous-programme fortran	24 si 0 paramètre, 29 + 12p sinon	65
135	DECOD	Décodeur	6	6
136	CYCLE1	Premier passage dans la routine cycle	73	73
137	CYCLE2	Passages suivants dans la routine cycle, cas d'une demande de plusieurs cycles	43	43
138	CYCLEFIN	Retour au superviseur après exécution du dernier cycle	12	12